Universidade Federal de Santa Catarina
Departamento de Engenharia Elétrica
Laboratório de Controle e Microinformática

Introdução à Linguagem Pascal

Prof. Carlos Maziero

Versão 1.2

Florianópolis, fevereiro de 1997

Resumo

Este texto contém uma introdução à programação de computadores usando como base a linguagem Pascal. São abordados os principais temas relativos à metodologia básica de programação e às características básicas da linguagem Pascal, todavia sem a pretensão de ser completo em nenhum destes temas.

Este texto foi inicialmente elaborado para a disciplina de introdução à Informática do curso de Engenharia de Controle e Automação da UFSC, e posteriormente revisada e aumentada.

Podem ser usados como textos de apoio os livros citados na bibliografia. Recomendo em particular [GL85, MM89, FE93] para os conceitos básicos de programação e exercícios complementares, e [O'B92] como guia de consulta para o compilador Turbo-Pascal.

A elaboração deste documento foi possível graças à participação ativa do CAECA - Centro Acadêmico de Engenharia de Controle e Automação, em particular os alunos Alexandre Orth, Gustavo Puchalski e Sérgio Hermesmeyer Jr.

Sumário

Capítulo 1

Conceitos fundamentais

1.1 O que é informática ?

Podemos definir a informática como a "ciência do tratamento automático das informações". Muito mais que visar simplesmente a programação de computadores para executar tarefas específicas, a informática estuda a estrutura e o tratamento das informações sob suas mais variadas formas: números, textos, gráficos, imagens, sons, etc. O computador em si intervém apenas como um instrumento para agilizar o tratamento da informação, e não como seu objetivo final. A informática busca criar uma abtração da realidade dentro de um sistema de computação, com o objetivo de reproduzi-lo o mais fielmente possível e assim poder substitui-lo, ou melhorar sua compreensão. A informática tem importância fundamental em controle e automação, por exemplo no estudo dos sistemas de controle (simulação, etc.), controle de dispositivos (robôs, etc), monitoração de processos físicos, etc.

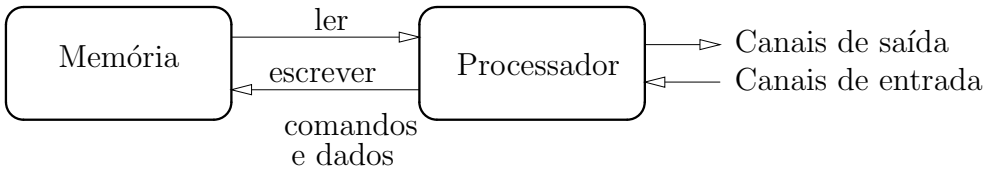
1.2 O computador

O computador é uma máquina capaz de receber, armazenar, tratar e produzir informações de forma automática, com grande rapidez e precisão. A evolução dos sistemas de computação teve seu início no século 16, mas estes somente mostraram-se úteis neste século, e sua vulgarização se deu graças à recente evolução na microeletrônica. Eis um breve resumo, bastante incompleto, dessa evolução:

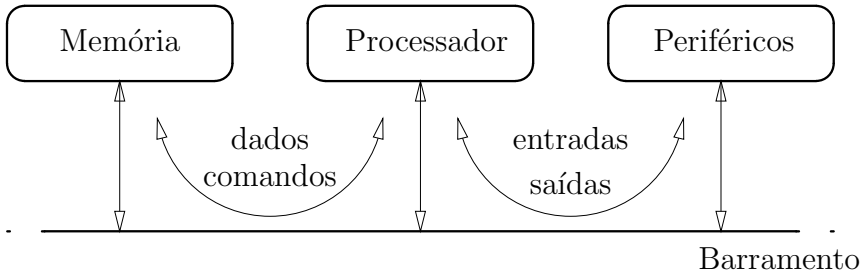
- *século 16*: Pascal e Leibniz propõe calculadoras baseadas em engrenagens.
- *século 19*: Charles Babbage constrói um computador programável mecânico.
- *década de 30*: computadores baseados em relés são usados para cálculos de balística.
- *1943*: construido o Eniac, com 18.000 válvulas.
- *1948*: invenção do transistor.
- *1951*: primeiro computador comercial, o Univac I.
- *anos 60*: apogeu dos computadores transistorizados.
- *anos 70*: circuitos integrados, invenção do micro-processador.
- *anos 80*: integração em larga escala (VLSI).

- *anos 90*: mais de 10^7 transitores em um chip.
- *futuro*: memórias biológicas; processadores usando luz; ...

A grande maioria dos computadores existentes atualmente segue um modelo proposto pelo matemático americano Von Neumann, por volta de 1940. Nesse modelo, um elemento processador segue as instruções armazenadas em uma memória de programas, para ler canais de entrada, enviar comandos sobre canais de saída e alterar as informações contidas em uma memória de dados. A figura abaixo indica a estrutura desse modelo:



Esse modelo inicial evoluiu para uma estrutura em barramento, que é a base dos computadores modernos. Nessa estrutura, as memórias de dados e de programa são fundidas em uma memória única, e as comunicações entre elementos são efetuadas através de uma via comum de alta velocidade:



Processador : unidade responsável pelo tratamento das informações. Nos computadores domésticos há geralmente uma única CPU (*central processing unit*). Máquinas mais potentes podem possuir dezenas, centenas ou milhares de processadores operando em paralelo (como a famosa *Connection Machine*, com 16K processadores).

Memória : onde são armazenados os programas que controlam o processador, e também as informações que estão sendo tratadas. Existem basicamente dois tipos de memórias:

ROM (Read-only memory) : contém dados pré-gravados que podem ser lidos pelo processador, mas não podem ser modificados. Serve geralmente para guardar os programas e informações necessárias no momento em que o computador é ligado, como configurações básicas de hardware, programas fixos, etc.

RAM (Random-acess memory) : pode ser lida e modificada pelo processador a qualquer momento. É usada para armazenar os programas em execução e seus dados.

Periféricos : são os dispositivos responsáveis pelas entradas e saídas de dados do computador, ou seja, pelas interações entre o computador e o mundo exterior. Como exemplos temos o teclado, o vídeo, as unidades de disco, o *mouse*, etc.

Barramento : é uma via de comunicação de alto desempenho, por onde circulam os dados tratados pelo computador.

Todos as informações contidas na memória do computador, bem como as comunicações entre seus diferentes elementos, são codificados sob a forma de sinais elétricos do tipo "ligado" e "desligado", representados pelos números 1 (ligado) e 0 (desligado). Esses sinais são chamados *bits* (*binary digits*). Como um bit representa uma quantidade de informações muito pequena, o computador trabalha com grupos de 8 bits, chamados *bytes*. Um byte representa um número binário, cujo valor é um inteiro positivo entre 0 e 255. Bytes podem ser agrupados para construir informações mais complexas, como números com sinal, reais, palavras, textos, imagens, sons, etc.

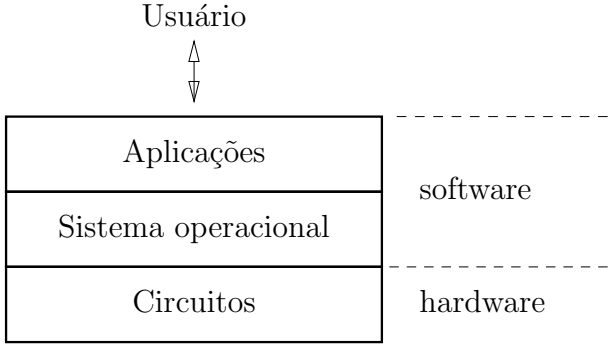
1.3 Uso dos computadores

Um sistema de computação é composto de uma parte física, constituída pelos circuitos e componentes que compõe o computador e seus periféricos, que chamamos *hardware*, e de uma parte lógica, composta pelos programas necessários para o funcionamento e uso do sistema, que chamamos *software*. Essa parte lógica normalmente pode ser dividida em dois níveis:

Sistema operacional : realiza a interface entre os programas do usuário e os circuitos do computador. Todas as ordens dadas ao *hardware* passam por ele: ler o teclado, produzir um som, mostrar algo na tela, imprimir um texto, etc. Exemplos de sistemas operacionais: MSDOS, Windows 95, UNIX, OS/2, etc.

Aplicativos : São os programas que interagem diretamente com o usuário: editores de texto, planilhas de cálculo, jogos, etc. Eles usam os serviços oferecidos pelo sistema operacional para atingir seu objetivo. Como exemplos temos o Word, Excel, Dbase, etc.

A figura abaixo indica a estruturação dos diferentes níveis de funcionalidade de um sistema de computação.



Essa estrutura em camadas tem importância fundamental porque permite esconder das aplicações (e portanto do usuário) a complexidade e diversidade do *hardware* que compõe a máquina. Desta forma, as aplicações podem ser construídas mais facilmente e com menor dependência de um hardware específico, pois o sistema operacional mascara as diferenças e oferece uma interface homogênea às mesmas.

Capítulo 2

A programação de computadores

Para programar um computador precisamos descrever exatamente o que queremos que ele execute, usando linguagens específicas para este fim. Devemos modelar a situação que queremos representar internamente no computador com todas as características e propriedades importantes, para que nossa implementação esteja correta e cumpra seus objetivos. Neste capítulo veremos como descrever as atividades a executar para cumprir uma determinada tarefa, de forma clara e estruturada.

2.1 Algoritmos

Uma ferramenta essencial para a construção de programas são os *algoritmos*. Um algoritmo é a especificação passo-a-passo das tarefas necessárias à resolução de um determinado problema. Exemplos de algoritmos seriam as receitas de cozinha, ou as instruções de montagem de um aparelho. Por exemplo, vejamos qual seria o algoritmo usado para trocar um pneu furado:

1. Pegar o macaco e o estepe no porta-malas do carro.
2. Levantar o carro usando o macaco.
3. Retirar o pneu furado.
4. Colocar o estepe em seu lugar.
5. Abaixar o carro.
6. Guardar o macaco e o pneu furado.

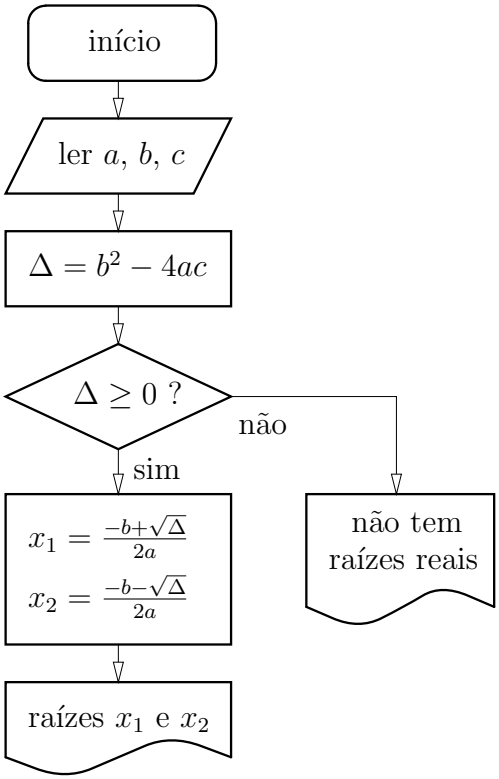
Esse passos devem ser detalhados até que o algoritmo represente completamente a situação que desejamos modelar, eliminando todas as dúvidas, imprecisões e ambigüidades. Por exemplo, a etapa 2 poderia ser refinada em:

2. Levantar o carro usando o macaco.
   2.1. Colocar o macaco sob o carro, próximo ao pneu a trocar.
   2.2. Girar a manivela do macaco até que o pneu se eleve do chão.

Para ser convertido em um programa de computador, um algoritmo deve ser descrito de forma clara e estruturada. Esse tipo de descrição ajuda inclusive na compreensão do algoritmo e na correção de eventuais erros. As duas formas de descrição de algoritmos mais simples e difundidas são os *fluxogramas* e os *pseudo-códigos*, que veremos a seguir.

2.1.1 Fluxogramas

Os fluxogramas, são representações gráficas dos algoritmos, construídas utilizando blocos para indicar as ações e decisões, e setas para indicar a seqüência de passos. Cada bloco tem uma forma diferente, que identifica sua função: entrada, saída, ação, decisão, etc. A figura abaixo mostra o fluxograma (simplificado) de um algoritmo para indicar as raízes de uma equação do segundo grau, tendo como entrada seus três coeficientes a, b e c:



2.1.2 Pseudo-código

Outra forma de descrever um algoritmo é usando construções similares às usadas nas linguagens de programação reais, por isso chamadas *pseudo-código*. Essa abordagem facilita mais tarde a programação do algoritmo assim especificado. A descrição de um algoritmo em pseudo-código se baseia em construções chamadas *estruturas básicas de controle*, que podem ser combinadas entre si. As estruturas básicas de controle mais empregadas são:

Alternativa : permite escolher entre duas ações diferentes, em função de uma condição dada. Sua estrutura segue a forma:

```
se condição então
   ação A
senão
   ação B
fim
```

A segunda parte (opção `senão` e *ação B*) pode ser omitida quando não for necessária. Um exemplo de uso dessa estrutura seria:

```
se X < 0 então
  escreva 'X não tem raiz real'
senão
  escreva a raiz de X
fim
```

Repetição : essa estrutura permite definir a repetição de uma ou mais ações até que uma determinada condição seja verdadeira:

```
repita
   ação A
   ação B
   ...
até que condição
```

Na estrutura acima, a condição somente é testada no final de cada iteração. Uma forma alternativa de estrutura de repetição permite testar uma condição dada no início de cada iteração; as ações designadas serão efetuadas enquanto a condição indicada for verdadeira:

```
enquanto condição faça
   ação A
   ação B
   ...
fim
```

Vejamos exemplos de uso dessas estruturas:

```
enquanto  parafuso está frouxo faça
   gire o parafuso no sentido horário
fim

repita
   mexa a colher
até que  massa esteja homogênea
```

Como vimos nos exemplos acima, as instruções englobadas por uma estrutura de controle são deslocadas levemente à direita em relação às linhas que definem a estrutura de controle propriamente dita. Esta técnica é chamada *indentação*, e ajuda a indicar as dependências entre instruções, permitindo assim uma melhor visualização do algoritmo e facilitando sua compreensão. Podem existir diversos níveis de indentação, no caso de estruturas de controle aninhadas (uma dentro da outra), como mostra o exemplo abaixo:

```
se condição 1 então
   ação A
   repita
      ação B
      ação C
   até que condição 2
senão
   ação D
fim
```

2.1.3 Passos para a construção de um bom algoritmo

1. Ler atentamente o enunciado para compreender o problema, analisando a situação a ser representada;
2. identificar as entradas e saídas de dados, ou seja, as informações a fornecer ao programa e os resultados que este deve retornar;
3. determinar o que deve ser feito para transformar as entradas recebidas nas saídas desejadas;
4. dividir o problema em suas partes principais (módulos), para facilitar a compreensão do todo (estratégia "*dividir para conquistar*");
5. analisar a divisão obtida para garantir sua coerência;
6. subdividir as partes mal compreendidas;
7. construir o algoritmo;
8. executar manualmente o algoritmo, para testa-lo.

2.2 Estruturas de dados

A matéria prima tratada pelo computador é a informação. As informações que compõe o mundo real podem ser armazenadas na memória do computador sob a forma de *estruturas de dados*. Podemos classificar as informações tratadas por um computador como sendo compostas por elementos pertencendo a um dos quatro tipos básicos de dados descritos abaixo (também chamados tipos primitivos ou básicos):

Inteiro : um dado de tipo inteiro é uma informação numérica pertencente ao conjunto dos inteiros Z. Alguns exemplos: `37` pessoas estão inscritas nesta turma. Ontem foram assaltados `4` bancos em Florianópolis.

Real : um dado deste tipo é uma informação numérica pertencente ao conjunto dos reais R. Alguns exemplos: Meu saldo bancário é de R$ `-217.43`. A média da turma foi `2.17` na última prova.

Caractere : é uma informação composta por uma letra ou seqüência de letras, dígitos e símbolos (também chamada *string*). Por exemplo: imprimir o histórico escolar de `José Antônio Neves Pontes`, cuja identidade é `6/R-1.543.433-SSP/SC`.

Lógico : informações deste tipo podem assumir somente um valor entre duas possibilidades: `verdadeiro` ou `falso`. Por exemplo: o aluno foi `aprovado` ou `reprovado`; a lâmpada está `acesa` ou `apagada`.

Os tipos básicos de dados podem variar de linguagem para linguagem, mas geralmente são similares aos descritos acima. A maioria das linguagens permite definir novos tipos de dados, a partir de seus tipos pré-definidos.

2.2.1 Variáveis

O computador usa a memória para armazenar os dados que está tratando. Podemos fazer uma analogia simples entre a memória do computador e um grande armário cheio de gavetas. Cada gaveta possui um nome e guarda um dado de um tipo determinado. Essas gavetas são chamadas *variáveis*, e cada uma pode conter um valor cujo tipo é definido no início do programa. No exemplo abaixo, a gaveta chamada `Aluno` possui o valor `Pedro Silva`, e assim por diante.

`Aluno:`	`Pedro Silva`	`Idade:`	`18`
`Peso:`	`76.0`	`Altura:`	`1.76`
`Casado:`	`sim`	`Filhos:`	`2`

Os nomes das variáveis devem obedecer a regras precisas para sua definição. Na maioria das linguagens de programação convencionais não é possível nem desejável identificar uma variável com algo do tipo "`nome do último colocado no concurso vestibular`". De modo geral, os nomes de variáveis podem conter letras e números, devem começar por uma letra e não podem conter símbolos especiais (com exceção do "_"). São nomes válidos: `Alpha`, `x17`, `Nota_Final`, `Média`. São nomes inválidos: `52Pst`, `E(s)`, `A:B`, `Nota-Final`, `X*`, `P%`, `Nota$`, ...

É muito importante que o nome usado para uma variável indique com clareza sua finalidade, para tornar o programa mais compreensível e portanto menos sujeito a erros de programação. Por exemplo, uma variável usada para armazenar o nome de um cliente em um programa deve ter um nome da forma `NomeCliente`, ou semelhante, e nunca somente `N` ou `NC`. Imagine um programa com 1000 linhas ou mais no qual a maioria da variáveis se chama `A`, `X`, `n`, etc.

No início de um programa de computador precisamos definir que variáveis iremos usar, e que tipo de dados podem ser armazenados nelas. Isso é efetuado através de uma *declaração de variáveis*. Vejamos um exemplo de declaração de variáveis em pseudo-código:

```
variáveis
   Aluno        : caractere [30]
   idade,filhos : inteiro
   altura, peso : real
   casado       : lógico
```

Neste exemplo, declaramos as variáveis `nome`, que pode conter um valor de tipo caractere (o indicador `[30]` indica que são aceitos valores com até 30 caracteres); as variáveis `idade` e `filhos`, que podem armazenar valores inteiros, as variáveis `altura` e `peso`, que podem conter valores de tipo real, e a variável `casado`, que pode armazenar um dado de tipo lógico (`verdadeiro` ou `falso`).

Uma vez declaradas as variáveis, podemos usá-las em programas, para armazenar os valores que serão consultados e/ou manipulados durante a execução dos mesmos. Para utilizar o valor de uma variável basta indicar seu nome onde desejado. Para armazenar um valor em uma variável utilizamos um *comando de atribuição*, que nos permite fornecer um valor a uma variável, ou seja, "guardar uma informação em uma gaveta". O tipo desse valor deve ser compatível com o tipo declarado para a variável. O comando de atribuição tem a seguinte forma:

variável `:=` *expressão do mesmo tipo da variável*

A expressão à direita do sinal "`:=`" é resolvida primeiro, e seu valor é em seguida atribuído à variável da esquerda do sinal, que deve ser do mesmo tipo resultante da expressão. Vejamos um exemplo em pseudo-código:

```
variáveis
  Número      : inteiro
  Soma, Média : real
  Aprovou     : lógico

início
  Soma    := 246,34
  Número  := 37
  Média   := Soma / Número
  Aprovou := (Média > 5.0)
fim
```

2.3 Constantes

Em um programa também podemos declarar informações constantes, que não devem mudar ao longo do programa. É o caso de constantes matemáticas, por exemplo, ou de informações como o nome do programador, a versão do programa, etc. As constantes podem ser declaradas em pseudo-código de forma semelhante às variáveis, com a inclusão de seu valor (fixo):

```
constantes
  pi          = 3.141592653589793264
  Versão      = '1.3b'
  Programador = 'Mickey Mouse'
```

As constantes podem ser usadas da mesma forma que as variáveis, mas seu valor não pode ser modificado (ou seja, uma constante nunca pode aparecer no lado esquerdo de uma atribuição).

2.4 Expressões e operadores

As expressões permitem combinar variáveis, constantes e operadores, para obter novos valores que podem ser usados nos algoritmos. Temos basicamente três tipos de operadores: aritméticos, relacionais e lógicos:

2.4.1 Operadores aritméticos

Relacionam entre si valores ou expressões numéricas inteiras ou reais, dando como resultado valores numéricos (inteiros ou reais). Por exemplo, se `x` for uma variável de tipo numérico, então

$$\frac{x+3}{x^2-3x+1}$$

é uma expressão empregando operadores aritméticos e que resulta em um valor numérico. Os operadores numéricos mais usuais em informática, e sua representação em pseudo-código são:

soma	`x + y`
subtração	`x - y`
produto	`x * y`
divisão real	`x / y`
potência	`x ^ y`
divisão inteira	`x div y`
resto da divisão	`x mod y`

Além dos operadores aritméticos, a maioria das linguagens de programação oferece um vasto conjunto de funções matemáticas, necessárias para cálculos de maior complexidade. As funções mais usuais são:

`sin(x)`	seno de x (em radianos)
`cos(x)`	cosseno de x
`tg(x)`	tangente de x
`arcsen(x)`	arco-seno de x
`arccos(x)`	arco-cosseno de x
`abs(x)`	valor absoluto de x
`int(x)`	parte inteira de x
`frac(x)`	parte fracionária de x
`random(x)`	valor aleatório inteiro entre 0 e x

2.4.2 Operadores relacionais

Estes operadores relacionam expressões numéricas entre si e dão como resultado valores lógicos. Por exemplo, a expressão

$$x + 3 \geq 7$$

resulta em `verdade` quando $x \geq 4$ e `falso` caso contrário. Os principais operadores relacionais são $> < = \neq \leq$ e $\geq$.

2.4.3 Operadores lógicos

Este operadores relacionam entre si valores ou expressões lógicas, resultando em valores lógicos. Os mais usuais são:

`NÃO` : nega ou inverte o resultado de uma expressão. Por exemplo, se `x` $\geq$ `17` é verdade, então `NÃO (x` $\geq$ `17)` é falso.

`E` : resulta em verdade somente se ambas as expressões forem verdadeiras. Por exemplo: `(x>5) E (x<10)` só será verdade se ambas as condições forem verdadeiras.

`OU` : resulta em verdade se ao menos uma das expressões for verdadeira. Por exemplo: `(x>5) OU (x<10)` será verdade se qualquer uma das duas condições for verdadeira, ou ambas.

2.4.4 Prioridades e parênteses

As prioridades usadas na resolução de expressões lógicas, aritméticas e relacionais são geralmente aquelas observadas na matemática, ou seja, resolvem-se nesta ordem: potenciações, multiplicações e divisões, somas e subtrações, operadores relacionais e operadores lógicos. Veja a resolução da expressão $(3^2 - 5 \geq 0)$ `E` $(\frac{5}{2} = 3 - 4)$ abaixo:

```
3^2 - 5 > 0  E  5 / 2 = 3 - 4
9   - 5 > 0  E  2.5   = -1
4       > 0  E  falso
verdade      E  falso
falso
```

Em caso de dúvidas quanto às prioridades, use e abuse dos parênteses !

2.5 Entrada e saída

De nada vale um computador efetuar cálculos e operações complexas se ele não puder receber dados e emitir resultados. Para receber dados do mundo exterior, o computador usa um comando de leitura de dados, representado em pseudo-código por `leia(`*variável 1*`,` *variável 2*`, ...)`. Este comando espera que o usuário digite um valor para cada uma das variáveis mencionadas entre parênteses: por exemplo, se `x1`, `x2` e `x3` são três variáveis declaradas de tipo `real`, o comando `leia (x1, x2, x3)` vai esperar que o usuário digite três valores reais e irá armazenar esses valores respectivamente nas variáveis `x1`, `x2` e `x3`. Outro exemplo:

```
nome         : caractere
altura, peso : real
idade        : inteiro

leia (nome, idade, peso, altura) ;
```

Para emitir resultados, o computador usa um comando de saída representado em pseudo-código por `escreva`, que possui a seguinte sintaxe: `escreva (`*expressão 1*`,` *expressão 2*`, ...)`. Esse comando permite apresentar na saída do computador (geralmente a tela ou a impressora), os resultados das expressões indicadas entre parênteses. Por exemplo, vejamos o que produziria a seqüência de comandos abaixo:

```
variáveis
  a,b: inteiro

início
  leia (a,b)
  se a > b então
    escreva (a,' é maior que ', b)
  senão
    se b > a então
      escreva (b,' é maior que ', a)
    senão
      escreva (a,' é igual a ', b)
    fim
  fim
fim
```

2.6 Exercícios

1. Você tem uma situação do mundo real a modelar sob a forma de um algoritmo: ir de ônibus da universidade à rodoviária. Construa o algoritmo aplicando os passos estudados neste capítulo.

2. Escreva um algoritmo para ler os três lados A, B e C de um triângulo e classificá-lo em equilátero, isósceles ou escaleno, em pseudo-código.

3. Desenhe o fluxograma do algoritmo para ir da Trindade ao centro de ônibus.

4. Desenhe o fluxograma do algoritmo proposto para a classificação de triângulos.

5. Especifique o algoritmo de um jogo no qual o computador sorteia um número entre 0 e 100 e o usuário tenta adivinhar, baseado somente nas dicas "é menor" ou "é maior". O jogo termina quando o usuário responde corretamente.

6. Construa um algoritmo para jogar o jogo da velha, usando as estruturas de controle vistas até o momento.

7. Você possui um robô que aceita os seguintes comandos:

   - `pegue` *objeto*
   - `pressione` *objeto*
   - `gire garra` (*ângulo positivo ou negativo*)
   - `mova` *objeto* `para` *lugar*
   - `desloque-se para` *lugar*

   Ele também é capaz de perceber quando um comando não pode mais ser executado. Que seqüência de ordens você daria ao robô para trocar uma lâmpada ?

8. Descreva as possíveis seqüências de ações para o algorítmo abaixo, considerando todos os valores possíveis para as condições dadas:

```
início
   ação A
   se condição 1 então
      ação D
      enquanto condição 2 faça
         ação E
      fim
      ação F
   senão
      repita
          ação B
      até que condição 2
      ação C
   fim
   ação A
fim
```

9. Todos os exercícios propostos no final do capítulo 2 do livro [GL85] podem ser resolvidos sem maiores dificuldades (com exceção dos exercícios 24b, 25 e 26b, que usam os diagramas de Chapin, não abordados neste texto).

Capítulo 3

Introdução à linguagem Pascal

3.1 Introdução

Pascal é uma linguagem de programação proposta no início dos anos 70 por Niklaus Wirth, no Instituto de Informática de Zurich, na Suíça. Suas principais características são:

- Simplicidade: trata-se de uma linguagem bem estruturada, organizada e facilmente compreensível, excelente para o ensino.
- Poder: embora simples, Pascal oferece estruturas de programação poderosas, que fazem dela uma linguagem bastante utilizada no desenvolvimento de aplicações.

3.2 Estrutura de um programa

Um programa em Pascal é composto por um cabeçalho, definido pela palavra reservada `program`, declarações de constantes, novos tipos, variáveis e sub-programas e um corpo, também chamado *programa principal*, delimitado pelas palavras reservadas `begin` e `end.`. O exemplo abaixo ilustra essa estrutura:

```
program nome do programa (input, output) ;
uses lista de bibliotecas a usar
  declaração de constantes
  declaração de novos tipos
  declaração de variáveis
  declaração de sub-programas
begin
  comandos do programa principal
end.
```

A declaração `uses`, que será vista com mais detalhes na seção 5.5, permite a inclusão de bibliotecas para o uso de diferentes recursos do sistema. Por enquanto nos limitaremos a indicar a biblioteca `crt`, que oferece as funções básicas de entrada e saída.

Nas versões mais recentes de Pascal a cláusula `(input, output)` é opcional, e por isso vamos omiti-la neste texto. Vejamos como ficaria o programa de classificação de triângulos escrito em Pascal:

```
program triangulos ;
uses crt ;
{ Este programa le os lados de um triangulo e classifica-o
  em equilatero, isosceles ou escaleno. }

var
  a,b,c : real ;

begin
  { ler os valores dos lados }
  readln (a,b,c) ;

  { testar equilatero }
  if (a = b) and (a = c) then
    writeln ('Triangulo equilatero')
  else
    { testar isosceles }
    if (a = b) or (a = c) or (b = c) then
      writeln ('Triangulo isosceles')
    else
      writeln ('Triangulo escaleno') ;
end.
```

Todo o texto escrito entre chaves ({}) é considerado como comentário, e portanto é ignorado pelo compilador. Comentários são muito importantes para a clareza do programa, e devem ser usados com abundância. Também é importante notar que a sintaxe de Pascal ignora espaços em branco e saltos de linha. Assim, a declaração de variáveis do programa acima poderia ter sido escrita da seguinte forma:

```
var a,b,c : real ;
```

Deve-se também observar que os comandos são geralmente terminados por um sinal de ponto-e-vírgula (`;`), mas não sempre. As regras de uso deste sinal serão melhor esclarecidas ao longo deste capítulo. Um último detalhe a observar é que normalmente os compiladores pascal não fazem distinção entre maiúsculas e minúsculas; desta forma podemos escrever `begin`, `BEGIN` ou até mesmo `BeGiN` !

3.3 Declaração de constantes e variáveis

Constantes são valores usados pelo programa, de qualquer tipo, que não devem mudar no decorrer de sua execução. Eis um exemplo de declaração de constantes:

```
const
  pi          = 3.141592653589793264 ;
  Versao      = '1.3b' ;
  Programador = 'Mickey Mouse' ;
```

Variáveis armazenam valores usados no programa, e que podem mudar no decorrer de sua execução. Os tipos básicos de variáveis definidos em Pascal são os seguintes:

- `integer`: inteiros com sinal, podendo ir de -32768 a +32767;

- `byte`: inteiros sem sinal, entre 0 e 255;
- `real`: reais, com 10 dígitos significativos, entre 10^{-38} e 10^{+38};
- `boolean`: lógicos, valendo `true` ou `false`;
- `char`: variável podendo armazenar um caractere;
- `string[n]`: cadeia com no máximo `n` caracteres; os caracteres e seqüências de caracteres devem ser sempre indicados entre aspas simples: `'a'`, `'um exemplo de string'`, etc;

Eis um exemplo de declaração de variáveis:

```
var
  i,j,k   : integer ;
  Final   : boolean ;
  Dia,Mês : byte ;
  Nota    : real ;
  Ano     : integer ;
  Nome    : string [40] ;
```

A atribuição de valores às variáveis segue a sintaxe apresentada anteriormente, através do operador "`:=`" :

```
i := 10 ;
j := j + 2 ;
Final := (j > i) ;
Nome := 'Zé Colméia' ;
Nota := 4.75 ;
```

3.4 Expressões e funções pré-definidas

Expressões aritméticas : podem envolver os tipos `integer`, `byte` e `real`, usando os operadores `+`, `-` , `*`, `/`, `div` e `mod`, e também parênteses. Não existem operadores para potência e raiz, mas existem funções pré-definidas para tal.

Expressões relacionais : envolvem tipos numéricos entre si, retornando os valores lógicos `true` ou `false`. Os seguintes operadores relacionais são definidos em Pascal: igual (`=`), diferente (`<>`), maior (`>`), menor (`<`), maior ou igual (`>=`) e menor ou igual (`<=`). Esses operadores permitem também comparar caracteres ou cadeias de caracteres, em termos de ordem alfabética (por exemplo, `'pedro' < 'paulo'` resulta em `false`).

Expressões lógicas : envolvem tipos lógicos entre si (ou resultados de expressões relacionais ou lógicas), usando os operadores `and`, `or` , `not` e `xor` (chamado *ou-exclusivo*: `a xor b` vale `true` somente se `a` for `true` e `b` for `false`, ou vice-versa). A tabela abaixo sitetiza o comportamento dos operadores lógicos envolvendo duas variáveis lógicas `A` e `B` (`T` indica `true` e `F` indica `false`):

A	B	not A	not B	A and B	A or B	A xor B
F	F	T	T	F	F	F
F	T	T	F	F	T	T
T	F	F	T	F	T	T
T	T	F	F	T	T	F

Funções pré-definidas : implementam operações diversas sobre valores numéricos. As funções mais usuais são (existem muitas outras):

`abs(x)`	valor absoluto de `x` (sem o sinal)
`int(x)`	parte inteira de `x`
`frac(x)`	parte fracionária de `x`
`round(x)`	valor arredondado de `x`
`ln(x)`	logarítmo natural de `x`
`exp(x)`	exponencial de `x` (e^x, com $e = 2.718...$)
`sqr(x)`	quadrado de `x`
`sqrt(x)`	raiz quadrada de `x`
`sin(x)`	seno de `x` (`x` em radianos)
`cos(x)`	cosseno de `x`
`tan(x)`	tangente de `x`
`arcsin(x)`	arco-seno de `x` (arco cujo seno é `x`)
`random(x)`	valor real aleatório entre 0 e `x`

3.5 Entrada e saída

Os comandos normalmente utilizados para entrada e saída de dados em Pascal são `read` e `write`, com suas variantes `readln` e `writeln`:

`read(var1,var2,...)` : permite ler valores (normalmente do teclado) e associá-los respectivamente às variáveis `var1`, `var2`, etc;

`readln(var1,var2,...)` : similar à anterior, mas move o cursor à linha seguinte imediatamente após a leitura, ignorando os valores ainda não lidos na linha atual. Assim, as próximas leituras ou escritas começarão na linha seguinte.

`write(expr1,expr2,...)` : permite escrever valores (normalmente na tela). Os valores escritos são os resultados das expressões numéricas, alfanuméricas ou lógicas entre parênteses.

`writeln(expr1,expr2,...)` : similar à anterior, mas move o cursor à linha seguinte imediatamente após a escrita. Assim, as próximas leituras ou escritas começarão em uma nova linha.

Por exemplo, o programa abaixo:

```
program leitura_escrita ;
uses crt ;
var
  i,j  : integer ;
  x    : real ;
  nome : string[20] ;

begin
  i := -342 ;
  j := 71 ;
  x := -3452.322 ;
  nome := 'Ambrosio Furtado' ;
  write ('i vale', i) ;
  writeln (' e j vale', j) ;
  write ('enquanto x vale', x, ' e o nome seria',nome) ;
end.
```

Produzirá a seguinte saída:

```
i vale        -342 e j vale          71
enquanto x vale-3.45232E+03 e o nome seriaAmbrosio Furtado
```

A saída de um programa Pascal é formatada de forma padrão, o que nem sempre permite obter os resultados na forma desejada. No entanto, os valores escritos por `write` e `writeln` podem ser formatados para melhor apresentação. Isto é obtido da forma `writeln (expressão:dimensão:decimais)`, onde `expressão` é o valor a ser escrito, `dimensão` indica o número total de casas a escrever (para inteiros, reais, lógicos e caracteres) e `decimais` representa o número de casas decimais a utilizar (somente no caso de expressões reais). Retomando o exemplo anterior, poderemos aplica a seguinte formatação:

```
  write ('i vale', i:5) ;
  writeln (' e j vale', j:3) ;
  write ('enquanto x vale', x:7:2, ' e o nome seria',nome:20) ;
```

O que nos produzirá a seguinte saída:

```
i vale -342 e j vale 71
enquanto x vale-3452.32 e o nome seria    Ambrosio Furtado
```

3.6 Estruturas de controle

As estruturas de controle em Pascal são similares às estudadas em pseudo-código, com algumas extensões. O fim de uma estrutura de controle é definido pelo "`;`", por isso seu uso deve ser atentamente observado.

Bloco de comandos : permite agrupar vários comandos para tratá-los como um só. Seu uso é geralmente associado às demais estruturas de controle, pois estas consideram normalmente um único comando:

```
begin
  comando1 ;
  comando2 ;
  ...
end ;
```

Se-então-senão : testa o valor de uma condição (que deve ser uma expressão resultando um valor lógico) para decidir que comandos efetuar. Caso mais de um comando deva ser executado, estes devem compor um bloco. Esta estrutura possui uma variante simples e outra usando o `else` (note a supressão do ponto-e-vírgula antes do `else`):

```
if condição then
  comando ;

 ou

if condição then
   comando1
else
   comando2 ;
```

Vejamos seu uso no exemplo abaixo, que é um programa para o cálculo de raízes de equações do 2º grau (observe o uso do bloco, e lembre que os comentários não são considerados comandos):

```
program raizes ;
uses crt ;
var
  a, b, c, x1, x2, det : real ;

begin
  readln (a,b,c) ;
  det := sqr(b) - 4*a*c ;
  if det >= 0 then
    begin
      { calcula e escreve as raízes reais }
      x1 := (-b + sqrt(det)) / (2*a) ;
      x2 := (-b - sqrt(det)) / (2*a) ;
      writeln ('As raizes sao ', x1,' e ', x2) ;
    end
  else
    { erro: somente raízes complexas }
    writeln ('A equacao nao possui raizes reais') ;
end.
```

Enquanto-faça : repete a instrução (ou o bloco de instruções) enquanto a condição dada for verdadeira. A condição é testada na entrada do comando:

```
while condição do
   comando ;
```

Por exemplo, vamos determinar os múltiplos de 2 inferiores a 1000:

```
program multiplos ;
uses crt ;
var
  i : integer ;

begin
  i := 1 ;
  while ( i < 1000 ) do
    begin
      write (i:5) ;
      i := i * 2 ;
    end ;
end.
```

O trecho de programa acima produz o seguinte resultado:

```
    1    2    4    8   16   32   64  128  256  512
```

Repita-até : repete as instruções até que a condição dada seja verdadeira. A condição é testada na saída do comando, e não há necessidade de se agrupar as instruções a repetir em um bloco:

```
repeat
  comando1 ;
  comando2 ;
  ...
until condição ;
```

Usando essa estrutura, vamos escrever um trecho de programa para ler valores digitados pelo usuário e somá-los. até receber um zero, quando então a soma deve ser apresentada:

```
program repita ;
uses crt ;
var
  valor, soma : integer ;

begin
  soma := 0 ;
  repeat
    readln (valor) ;
    soma := soma + valor ;
  until valor = 0 ;
  writeln ('A soma vale ', soma) ;
end.
```

Para-faça : permite efetuar ciclos incondicionais, usando uma variável de controle. A variável recebe inicialmente o valor `valor1` e após cada ciclo é automaticamente incrementada (+1) (ou decrementada, no caso da opção `downto`). A repetição termina quando a variável de controle ultrapassar `valor2`. Pode-se contar de forma crescente ou decrescente:

```
for variável := valor1 to valor2 do
  comando ;

 ou

for variável := valor1 downto valor2 do
  comando ;
```

Por exemplo, programa abaixo calcula a soma do 20 primeiros inteiros positivos e de seus quadrados:

```
program quadrados ;
uses crt ;
var
  soma, soma2,i : integer ;

begin
  soma := 0 ;
  soma2 := 0 ;
  for i := 1 to 20 do
    begin
      soma := soma + i ;
      soma2 := soma2 + sqr(i) ;
    end ;
  writeln ('A soma dos numeros   vale ', soma) ;
  writeln ('A soma dos quadrados vale ', soma2) ;
end.
```

O programa acima nos produz o seguinte resultado:

```
A soma dos numeros   vale          210
A soma dos quadrados vale         2870
```

É importante observar que as instruções efetuadas no interior do ciclo não devem em hipótese alguma modificar o valor da variável de controle, sob pena de dificultar a compreensão do fluxo do programa.

Caso-faça : permite escolher os comandos a efetuar em função do resultado de uma expressão. A expressão e os valores testados devem ser de mesmo tipo, inteiro ou caractere:

```
case expressão of
  valor1, valor2, ... : comandoA ;
  valorx, valory, ... : comandoB ;
  ...
else
   comandoZ ;
end ;
```

Por exemplo, o trecho de programa abaixo escreve os dias da semana a partir de um código numérico do dia, entre 1 e 7, e atualiza um contador de domingos:

```
case CodigoDia of
  1 : begin
        writeln ('domingo') ;
        ContDomingos := ContDomingos + 1 ;
      end ;
  2 : writeln ('segunda') ;
  3 : writeln ('terça') ;
  4 : writeln ('quarta') ;
  5 : writeln ('quinta') ;
  6 : writeln ('sexta') ;
  7 : writeln ('sábado') ;
else
  writeln ('código incorreto !');
end ;
```

3.7 Exercícios

Para a resolução dos exercícios a seguir, primeiro analise-os com atenção e use os passos indicados anteriormente para a construção dos algoritmos de resolução. Só então proceda à codificação dos algoritmos em Pascal.

1. Escreva um programa Pascal para o joguinho de adivinhação de números proposto anteriormente.

2. Escreva um programa Pascal que leia três números inteiros e os escreva em ordem crescente.

3. Construa um programa Pascal que calcule a parte inteira da raiz quadrada de um número dado, usando aproximações sucessivas (ou seja, sem usar funções matemáticas). Depois melhore esse algoritmo para calcular também a parte fracionária, com um determinado número de casas decimais.

4. Escreva um programa Pascal para determinar a média, o valor mínimo e o valor máximo de uma lista de n números.

5. Escreva um programa Pascal que calcule $n!$ (fatorial de n).

6. Construa um programa Pascal para gerar o gráfico da função $y = e^{-at}\sin(0.5t)$. Sugestão: faça cada linha da saída corresponder a um valor de t, iniciando em $t = 0$; determine o valor $y(t)$ da função para a linha e escreva um "*" precedido por y espaços em branco, como mostra o exemplo abaixo:

```
--------------------------------*----------------------------------
                                |     *
                                |          *
                                |             *
                                |              *
                                |             *
                                :
```

7. Construa um programa em Pascal para calcular os n primeiros números primos.

8. Construa um programa Pascal para o cálculo do imposto de renda de um grupo de contribuintes, considerando que os dados de cada contribuinte (nome, CPF, renda mensal e número de dependentes) serão digitados pelo usuário. Do imposto devido deve ser deduzido um desconto de 5% por dependente; as alíquotas para o cálculo do imposto são:

Renda líquida	Alíquota
até 2 SM	isento
até 3 SM	5%
até 5 SM	10%
até 7 SM	15%
acima de 7 SM	20%

   O último contribuinte, que não deve ser considerado, terá seu CPF igual a zero. O valor do salário mínimo deve ser lido no início do programa.

9. Gere a seguinte pirâmide de dígitos, usando laços para-faça (procure entender a regra de construção da pirâmide; não se limite a somente escrever 10 cadeias de caracteres):

```
                1
               232
              34543
             4567654
            567898765
           67890109876
          7890123210987
         890123454321098
        90123456765432109
       0123456789876543210
```

10. Escreva um programa que leia um número inteiro positivo e escreva seu equivalente em algarismos romanos. O programa deve ler e escrever valores até receber um zero como entrada.

Capítulo 4

Estruturas de dados

Nem sempre os tipos básicos de dados conhecidos, apresentados no capítulo anterior, tem a flexibilidade necessária para permitir a construção de programas complexos de maneira eficiente. Por exemplo, construir um programa para dispor em ordem crescente as notas de 1000 alunos usando somente variáveis dos tipos básicos seria completamente inviável. As linguagens de programação oferecem estruturas de dados específicas para esse fim, como os vetores, matrizes e registros.

4.1 Vetores

O problema das notas de alunos apresentado acima pode ser facilmente resolvido em Pascal usando-se uma estrutura de dados homogênea unidimensional chamada *vetor*. Um vetor é uma estrutura de dados contendo um número pré-definido de elementos de um mesmo tipo, e que podem ser acessados individualmente, através de um índice variável. O vetor usado na solução do problema acima teria a seguinte forma:

Índice	1	2	3	4	5	6	...	999	1000
Valor	3.7	7.8	9.6	8.9	4.9	6.1	...	7.4	8.4

Esta estrutura, com capacidade para armazenar até 1000 elementos de tipo real, pode ser facilmente definida em Pascal. Para defini-la usamos a declaração `type`:

```
type
  VetorReal = array [1..1000] of real ;
```

A declaração `type` permite a definição de novos tipos de dados em Pascal. No exemplo acima definimos um novo tipo, chamado `VetorReal`, que pode conter 1000 valores de tipo `real`. Observe que a declaração `type` não cria a estrutura de dados em memória, mas apenas define a sua forma. Esse novo tipo pode então ser usado para a criação de variáveis, exatamente como fazemos para os tipos básicos:

```
var
  VetorNotas : VetorReal ;
```

O acesso a um elemento de um vetor é efetuado através de seu índice, que deve ser indicado entre colchetes (`[ ]`) após o nome da variável. Como índice podemos usar qualquer expressão

inteira que retorne um valor dentro dos limites estabelecidos (`[1..1000]`). Assim, o acesso aos elementos da variável `VetorNotas` se efetua da seguinte forma (`i` e `k` são variáveis do tipo `integer`):

```
VetorNotas[17] := 6.4 ;

if VetorNotas[k] > 5 then
  Aprovados := Aprovados + 1 ;

for i := 1 to 1000 do
  VetorNotas[i] := 0.0 ;
```

Vejamos um exemplo do uso de vetores: desejamos calcular a média das notas de uma turma, e indicar o percentual de alunos que tiraram notas acima da média. Os passos necessários para cumprir esta tarefa são:

1. O primeiro passo é ler as notas. Para isso, primeiro precisamos saber quantas notas existem, e depois efetuar a leitura de cada uma:

   ```
   readln (NumNotas) ;
   for i := 1 to NumNotas do
     readln (VetorNotas[i]) ;
   ```

2. A seguir podemos proceder ao cálculo da média: precisamos somar todas as notas e depois dividir a soma pelo número de notas:

   ```
   Soma := 0.0 ;
   for i := 1 to NumNotas do
     Soma := Soma + VetorNotas[i] ;
   Media := Soma / NumNotas ;
   ```

3. Agora que obtivemos a média, devemos percorrer novamente as notas, contando quantas estão acima da média:

   ```
   Acima := 0 ;
   for i := 1 to NumNotas do
     if VetorNotas[i] > Media then
       Acima := Acima + 1 ;
   Porcent := 100 * Acima / NumNotas ;
   ```

4. Finalmente, podemos escrever os resultados:

   ```
   writeln ('A media das ', NumNotas:4,' notas é ', Média:5:2) ;
   writeln ('Acima da media: ', Porcent:5:2,'%') ;
   ```

Juntando aos trechos de código acima as declarações de variáveis e tipos necessárias, teremos um programa Pascal completo.

Vejamos outro exemplo: um algoritmo conhecido de ordenação de valores em ordem crescente ou decrescente é o chamado "método da bolha". Dado um vetor de valores V com n elementos, o método consiste em percorrer o vetor do início ao fim, comparando valores vizinhos V_i e V_{i+1} e permutando-os se $V_i > V_{i+1}$. O procedimento de varredura deve ser repetido até que não hajam mais trocas possíveis. Desta forma os valores maiores vão sendo lentamente deslocados para a direita e os menores para a esquerda (como bolhas que sobem num líquido), ordenando o vetor. Veja o exemplo abaixo, para um vetor inicial `V = [ 4 7 1 9 3 ]`:

Passo	Ação	Vetor resultante
0	valor inicial do vetor	`V = [ 4 7 1 9 3 ]`
1	permutar o par (7, 1)	`V = [ 4 1 7 9 3 ]`
2	permutar o par (9, 3)	`V = [ 4 1 7 3 9 ]`
3	permutar o par (4, 1)	`V = [ 1 4 7 3 9 ]`
4	permutar o par (7, 3)	`V = [ 1 4 3 7 9 ]`
5	permutar o par (4, 3)	`V = [ 1 3 4 7 9 ]`

A programação deste algoritmo em Pascal é bastante simples. Vejamos em pseudo-código a estrutura da solução:

```
inicio
  ler (vetor)
  repita
    varrer o vetor, efetuando as trocas possíveis
  ate que última varredura não teve trocas
  escrever (vetor)
fim
```

Como os procedimentos de leitura e escrita do vetor já foram vistos antes, vamos nos concentrar no laço central. Para varrer o vetor precisamos de um laço `para-faça`:

```
para i de 1 a n-1 faça
  se V[i] > V[i+1] então
    permutar V[i] com V[i+1]
  fim
fim
```

Para permutar os valores de `V[i]` e `V[i+1]` precisamos de uma variável auxiliar:

```
Auxiliar := V[i] ;
V[i]     := V[i+1] ;
V[i+1]   := Auxiliar ;
```

A varredura do vetor deve ser repetida até não haver mais trocas possíveis, ou seja, até que a última varredura realizada não tenha efetuado nenhuma troca. Para isso vamos precisar de uma variável lógica, que indique se houve troca na última varredura ou não:

```
repita
  HouveTroca := falso
  para i de 1 a n-1 faça
    se V[i] > V[i+1] então
      permutar V[i] com V[i+1]
      HouveTroca := verdade
    fim
  fim
ate que não HouveTroca
```

Com estes dados já nos é possível escrever o programa Pascal:

```
program bolha ;
uses crt ;

type
  VetorReal = array [1..1000] of real ;

var
  V   : VetorReal ;
  i,n : integer ;
  Auxiliar   : real ;
  HouveTroca : boolean ;

begin
  readln (n) ;                        { leitura do vetor }
  for i := 1 to n do
    readln (V[i]) ;

  repeat                              { ordenar o vetor }
    HouveTroca := false ;
    for i := 1 to n-1 do
      if V[i] > V[i+1] then
        begin
          Auxiliar := V[i] ;          { trocar V[i] com V[i+1] }
          V[i]     := V[i+1] ;
          V[i+1]   := Auxiliar ;
          HouveTroca := true ;
        end ;
  until not HouveTroca ;

  writeln ('Vetor ordenado:') ;       { escrever resultado }
  for i := 1 to n do
    writeln (V[i]:10:4) ;
end.
```

4.2 Matrizes

De forma geral, a declaração `array` (arranjo) pode ser utilizada para definir matrizes n-dimensionais de dados. O vetor é um caso particular dessa declaração, no qual a matriz definida possui somente uma dimensão (linha). A forma geral da definição de estruturas `array` é:

```
type
  nome = array [ inf1.. sup1,  inf2.. sup2,...] of tipo ;
```

Por exemplo, podemos definir uma matriz bidimensional de valores reais através das declarações Pascal que seguem:

```
type
  MatrizReal = array [1..4,1..6] of real ;

var
  Mat : MatrizReal ;
```

A variável `Mat` assim definida tem então 24 elementos:

Mat[1,1]	Mat[1,2]	Mat[1,3]	Mat[1,4]	Mat[1,5]	Mat[1,6]
Mat[2,1]	Mat[2,2]	Mat[2,3]	Mat[2,4]	Mat[2,5]	Mat[2,6]
Mat[3,1]	Mat[3,2]	Mat[3,3]	Mat[3,4]	Mat[3,5]	Mat[3,6]
Mat[4,1]	Mat[4,2]	Mat[4,3]	Mat[4,4]	Mat[4,5]	Mat[4,6]

O acesso aos elementos da matriz `Mat` se dá de maneira similar ao acesso aos elementos de um vetor, ou seja, através de seus índices. Eis um trecho de código em Pascal para ler as dimensões e os elementos da matriz `Mat` acima, com 4 linhas e 6 colunas:

```
for i := 1 to 4 do
  for j := 1 to 6 do
    readln (A[i,j]) ;
```

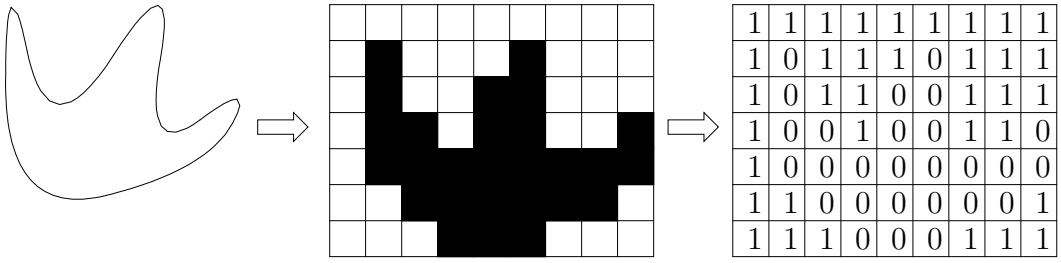
Observe que na estrutura acima o laço externo percorre as linhas, e o laço interno percorre as colunas. Assim, para cada linha percorremos todas as colunas.

As estruturas de dados vetoriais e matriciais são extremamente importantes em programas. Eis alguns exemplos de sua utilização:

- **Armazenar tabelas**: uma das utilizações mais freqüentes de matrizes é o armazenamento de tabelas de dados, como a tabela de dados atmosféricos abaixo:

hora	temperatura	umidade	pressão
00	12.5	70	778
01	12.1	67	785
02	11.4	65	789
...	...	...	...
23	13.0	73	770

- **Tratamento de imagens**: uma imagem em um computador é geralmente representada por uma matriz bidimensional, onde cada elemento e_{xy} representa um "quadradinho" (pixel) da imagem. Desta forma, cada elemento `e[x,y]` da matriz irá conter a cor e/ou tonalidade dessa região. No exemplo da figura a seguir, "0" indica a cor preta e "1" a cor branca:



1	1	1	1	1	1	1	1	1
1	0	1	1	1	0	1	1	1
1	0	1	1	0	0	1	1	1
1	0	0	1	0	0	1	1	0
1	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	1
1	1	1	0	0	0	1	1	1

- **Cálculo numérico**: matrizes são geralmente usadas para a representação e resolução de sistemas de equações lineares normalmente encontrados em problemas de engenharia:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ \cdots &= \cdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n \end{aligned}$$

4.3 Registros

Na seção anterior vimos com agrupar elementos de um mesmo tipo, usando vetores e matrizes. A estrutura chamada *registro* permite agrupar sob um mesmo nome dados de tipos distintos, o que é bastante útil em certas situações. Por exemplo, consideremos uma passagem de ônibus com as seguintes informações:

Data:	`14/11/96`	Número:	`3423`
Origem:	`Curitiba`	Destino	`Florianópolis`
Partida:	`13:30`	Chegada:	`18:00`
Assento:	`27`	Fumante:	`não`

Cada informação pertence a um tipo distinto, mas todas dizem respeito a uma determinada passagem de ônibus. Por isso, gostariamos de agrupá-las para poder tratá-las como uma só variável. Em Pascal pode-se definir uma estrutura de dados do tipo registro (`record`), contendo todas as informações necessárias para armazenar essa passagem de ônibus:

```
type
  Passagem = record
    Data            : string[8] ;
    Numero          : integer ;
    Origem,Destino  : string[20] ;
    Partida,Chegada : string[5] ;
    Assento         : byte ;
    Fumante         : boolean ;
  end ;
```

Para criar variáveis do tipo `Passagem`, procede-se como anteriormente:

```
var
  P1, P2 : Passagem ;
```

As variáveis `P1` e `P2` assim criadas são registros contendo cada uma todos os campos internos definidos para o tipo `Passagem`. A referência aos campos de um registro segue a forma `nome.campo`. Assim, para atribuirmos os valores do exemplo à variável `P1`, efetuamos:

```
P1.Data    := '14/11/96' ;
P1.Numero  := 453423 ;
P1.Origem  := 'Curitiba' ;
P1.Destino := 'Florianópolis' ;
P1.Partida := '13:30' ;
P1.Chegada := '18:00' ;
P1.Assento := 27 ;
P1.Fumante := false ;
```

O exemplo acima pode ser simplificado através do uso do operador `with`, que permite indicar uma única vez o registro cujos campos desejamos acessar:

```
with P1 do
  begin
    Data    := '14/11/96' ;
    Numero  := 453423 ;
    Origem  := 'Curitiba' ;
    Destino := 'Florianópolis' ;
    Partida := '13:30' ;
    Chegada := '18:00' ;
    Assento := 27 ;
    Fumante := false ;
  end ;
```

Podemos aninhar registros, ou seja, colocar registros dentro de registros, para melhor estruturar a informação. Por exemplo, poderíamos considerar as datas e horários da passagem não mais como strings mas como registros contendo campos internos. Vejamos como fica a definição dos tipos:

```
type
  TipoData = record
    dia, mes : byte ;
    ano      : integer ;
  end ;

  TipoHora = record
    hora, minuto : byte ;
  end ;

  Passagem = record
    Data            : TipoData ;
    Numero          : integer ;
    Origem,Destino  : string[20] ;
    Partida,Chegada : TipoHora ;
    Assento         : byte ;
    Fumante         : boolean ;
  end ;
```

O acesso aos campos segue a mesma regra definida anteriormente:

```
P1.Data.dia := 14 ;
P1.Data.mes := 11 ;
with P1.Partida do
  begin
    hora   := 13 ;
    minuto := 30 ;
  end ;
```

4.4 Combinando arranjos e registros

A utilidade dos registros torna-se evidente quando os utilizamos em conjunto com os arranjos. Podemos por exemplo definir um vetor contendo todas as passagens vendidas em um dia. A manipulação desse vetor deve seguir ao mesmo tempo as regras definidas para os arranjos e os registros:

```
var
  PassDia : array [1..1000] of Passagem ;

PassDia[4].Destino := 'Cuiabá';
```

Da mesma forma é possível declarar alguns dos campos internos de um registro como sendo arranjos. Veja como ficaria o registro de um aluno, contendo suas notas em 4 exames:

```
type
  FichaAluno = record
    Nome   : string[40];
    Matric : string[9];
    Nota   : array [1..4] of real;
  end;
```

Podemos em seguida definir um vetor contendo as fichas dos alunos de uma turma:

```
var
  Aluno: array [1..50] of FichaAluno;
```

O acesso à terceira nota do 26º aluno da turma ficaria assim:

```
Aluno[26].Nota[3] := 4.7 ;
```

4.5 Exercícios

Não se esqueça de analisar atentamente cada situação e construir os algoritmos considerando os passos enumerados anteriormente, antes de partir para a implementação.

1. Sendo um vetor `V = [2 6 8 3 10 16 1 21 33 14]` e as variáveis `x=2` e `y=4`, determine os valores correspondentes a:

```
V[x+1]                V[x+y]           V[V[V[7]]]
V[V[x*2] div V[4]]    V[2+V[8-V[2]]]   V[V[x+y]]
```

2. Construa um programa Pascal para calcular a matriz soma S_{mn} entre duas matrizes A_{mn} e B_{mn} (sugestão: $\forall(i,j)\ S_{ij} = A_{ij} + B_{ij}$).

3. Construa um programa Pascal para calcular o produto P_{mp} entre duas matrizes A_{mn} e B_{np} (sugestão: $\forall(i,j)\ P_{ij} = \sum_{k=1}^{n}(A_{ik} \cdot B_{kj})$).

4. Dada uma matriz de inteiros `m[0..1, 1..4, 1..3]`, cujo conteúdo é indicado pela figura a seguir, e um vetor de inteiros `v[1..4] = [3 1 3 0]`, determinar o valor dos elementos `m[1,1,2]`, `v[3]`, `v[m[0,1,1]]`, `m[v[4],v[2],v[1]]` e `m[m[v[4],4,3],v[m[0,3,1]],2]`.

	1	2	3	4
0	1	2	3	4
1	5	-5	3	0

1

	1	2	3	4
0	1	1	1	1
1	-3	2	0	0

2

	1	2	3	4
0	0	0	1	1
1	-1	-1	2	2

3

5. A distância entre várias cidades (em Km) é dada pela tabela abaixo:

	1	2	3	4	5
1	0	15	30	5	12
2	15	0	10	17	28
3	30	10	0	3	11
4	5	17	3	0	80
5	12	28	11	80	0

   Escreva programas em Pascal para:

   - Ler os valores de distância e montar a tabela. Procure otimizar a leitura, evitando ler duas vezes a mesma distância (por exemplo, distância 1-3 e distância 3-1).
   - Dadas duas cidades X e Y, escrever a distância entre elas.
   - Dado um trajeto com um número qualquer de cidades (terminando com uma cidade de número 0), informar a distância total percorrida.
   - Escrever a tabela de distância sem repetições (isto é, se a distância A-B foi escrita, não escrever a distância B-A).

6. Em computação gráfica é comum representarmos imagens através de matrizes. Seja uma matriz de inteiros $\mathcal{A}$, de dimensões 100×100, contendo uma imagem em níveis de cinza (como uma fotografia em preto-e-branco). O valores dos elementos $\mathcal{A}_{ij}$ se situam entre 0 e 100, formando uma escala crescente de tons cinza entre 0 (preto) e 100 (branco). Escreva um programa para calcular o negativo fotográfico de uma imagem assim representada. Dica: o negativo de 0 é 100, o de 1 é 99, o de 2 é 98 e assim por diante.

7. Dada uma matriz $\mathcal{M}$ de 4×5 elementos, escreva um programa para somar os elementos de cada linha, gerando um vetor de somas. Em seguida, some os elementos desse vetor para gerar a soma de todos os elementos da matriz, e imprima ambos os resultados.

8. Dados 2 vetores de inteiros `R[1..10]` e `S[1..20]`, construa um programa Pascal para:

   - Ler os dois vetores.
   - Construir um vetor `U` com a união (concatenação) de `R` e `S`.
   - Construir um vetor `C` com os valores comuns de `R` e `S`.
   - Construir um vetor `D` com os valores de `R` que não estão em `S`.

9. No "método da bolha", usado para ordenar vetores, podemos observar que após a primeira iteração o extremo direito do vetor contém o maior elemento, não necessitando ser percorrido novamente. Após a segunda iteração, o extremo direito contém os dois maiores elementos, e assim por diante. Modifique o programa apresentado para levar em conta essa observação.

10. Uma biblioteca possui obras de ciências exatas, humanas e biomédicas, num total de 1500 volumes (500 para cada área). Cada livro apresenta as seguintes informações: código, nome, autor, editora, número de páginas e se foi doado ou adquirido. Construa programas Pascal para:

    - Definir o registro de catálogo de cada livro, e um vetor de registros para cada área.
    - A partir do código, fornecido pelo usuário, apresentar o registro do livro (ou uma mensagem de erro, caso o código não corresponda a nenhum livro).
    - Ordenar um vetor de registros de livros em ordem crescente de nomes (atenção para mover os registros inteiros, e não somente os nomes).
    - Listar todas as obras de uma área que tenham sido doadas.
    - Efetuar a alteração de um registro: usuário fornece o código do livro e, se este existir, os novos valores para os demais campos.

11. Os demais exercícios dos capítulos 4, 5 e 6 do livro [GL85].

Capítulo 5

Programação modular

Um programa em Pascal é uma seqüência de instruções. À medida em que um programa cresce, ele pode se tornar complexo e pouco legível. Além disso, certas seqüências de comandos podem ser usadas com freqüência em diversos pontos do programa, tendo de ser inteiramente reescritas em cada um desses pontos, o que é certamente uma fonte de erros.

Para enfrentar essa situação podemos dividir nosso programa em módulos, separando logicamente as diferentes etapas do programa. Além disso, podemos agrupar trechos de código freqüentemente usados em um módulo separado, que pode ser ativado a partir de diversos pontos do programa. Em pascal isto pode ser efetuado através da definição de procedimentos, funções e bibliotecas, como veremos a seguir.

5.1 Procedimentos

Um procedimento é um sub-programa, ou seja, um módulo separado do programa principal, contendo comandos e que pode ser ativado a partir do programa principal, ou a partir de outros módulos. Normalmente um procedimento é composto por três elementos:

Interface : define o nome do procedimento e como o mesmo pode ser ativado. Ela também pode definir parâmetros, que são dados entregues ao procedimento, e resultados que este pode retornar a quem o ativou. Normalmente, a interface deve ser o único elo de ligação entre um procedimento e o restante do programa.

Contexto : Permite declarar variáveis internas ao procedimento, além de tipos, constantes e até mesmo outros procedimentos internos. As variáveis internas de um procedimento são chamadas *variáveis locais*, ao contrário das variáveis declaradas no programa principal, as *variáveis globais*, que são acessíveis a todos. O mesmo ocorre para os tipos, constantes e procedimentos locais.

Corpo : Define os comandos que serão executados pelo procedimento. Ao ser ativado, um procedimento efetua as operações definidas em seu corpo, e ao terminar a execução retorna ao ponto de onde foi ativado.

A forma geral de definição de um procedimento segue o modelo a seguir, no qual a interface é definida pela palavra `procedure`, o contexto pelas declarações locais e o corpo pelas palavras `begin` e `end`:

```
procedure nome ( param1 : tipo1 ;  param2 : tipo2 ; ...) ;
  declaração de constantes locais
  declaração de tipos locais
  declaração de variáveis locais
  declaração de procedimentos e funções locais
begin
  corpo do procedimento
end ;
```

Para exemplificar, vamos criar um procedimento simples, que escreva na tela uma linha de separação, composta por asteriscos. Sua forma mais simples, sem parâmetros nem declarações locais, seria:

```
procedure separador ;
begin
  writeln ('************************************************************') ;
end ;
```

Para ativar o procedimento acima bastar chamá-lo por seu nome:

```
begin { programa principal }
  ...
  separador ;
  ...
end.
```

5.1.1 Variáveis locais

Vamos modificar o procedimento `separador` definido acima, substituindo o comando de escrita simples por um laço do tipo *para-faça*. Precisamos declarar localmente uma variável para o controle do laço:

```
procedure separador ;
var
  i : integer ;

begin
  for i := 1 to 60 do
    write ('*') ;
  writeln ;
end ;
```

A variável `i` definida acima é uma variável local do procedimento `separador`, que só existirá dentro desse procedimento, deixando de existir quando ele termina.

Variáveis com o mesmo nome podem existir em diversos procedimentos, sem risco de interferência entre elas. Isto significa que a variável `i` definida no procedimento `separador` não será confundida com outras variáveis `i` definidas em outros procedimentos, ou mesmo no programa principal.

Além das variáveis, podem ser declaradas também constantes, tipos e até mesmo procedimentos e funções locais a um procedimento, e portanto desconhecidos fora deste.

5.1.2 Parâmetros

Se desejarmos passar informações a um procedimento, ou receber informações dele, devemos declarar parâmetros para o mesmo. Os dados declarados entre parênteses após o nome do procedimento constituem seus parâmetros, e podem ser usados no corpo do procedimento da mesma forma que uma variável local.

Por exemplo, vamos indicar ao procedimento `separador` o comprimento da linha a traçar e o caractere a ser usado na linha:

```
procedure separador (tamanho: integer ;  caract : char) ;
var
  i : integer ;

begin
  for i := 1 to tamanho do
    write (caract) ;
  writeln ;
end ;
```

Assim, para traçar uma linha com 45 asteriscos e outra com 70 cifrões, bastar chamar o procedimento `separador` com os parâmetros indicados, sem esquecer de obedecer a ordem de declaração:

```
begin { programa principal }
  separador (45, '*') ;
  separador (70, '$') ;
end.
```

Os parâmetros `tamanho` e `caract` definidos acima são *parâmetros de entrada* do procedimento `separador`. Eles podem ser modificados dentro do procedimento, sem que isso seja percebido fora do mesmo. Por exemplo, vamos incluir no procedimento acima testes para limitar o tamanho da linha traçada entre 1 e 80:

```
procedure separador (tamanho: integer ;  caract : char) ;
var
  i : integer ;

begin
  if tamanho < 1 then
    tamanho := 1 ;
  if tamanho > 80 then
    tamanho := 80 ;
  for i := 1 to tamanho do
    write (caract) ;
  writeln ;
end ;
```

Podemos também declarar parâmetros para retornar valores calculados dentro de um procedimento para o programa principal, ou para o procedimento que o ativou. Para isso usa-se o atributo `var` na declaração do parâmetro. Parâmetros assim declarados são considerados *parâmetros de entrada e saída*. Eles permitem passar dados ao procedimento e recebe-los de volta com as modificações efetuadas sobre o mesmo dentro do procedimento. Como exemplo, vejamos a definição de um procedimento `limitar (x)`, que limita o valor de `x` entre -100 e 100:

```
procedure limitar (var num : integer) ;
const
  limite = 100 ;

begin
  if num < -limite then
    num := -limite
  else
    if num > limite then
      num := limite ;
end ;
```

Para ativar o procedimento precisamos indicar como parâmetro de entrada e saída uma variável que irá conter o valor inicial e receberá o resultado. Todas as modificações efetuadas no parâmetro dentro do procedimento serão refletidas sobre a variável indicada, como mostra o exemplo abaixo.

```
begin
  x := 37 ;
  limitar (x) ;
  writeln (x) ;   { var escrever 37}
  x := -120 ;
  limitar (x) ;   { vai escrever -100 }
  writeln (x) ;
  limitar (3*x) ; { erro ! }
end.
```

A última chamada ao procedimento é inválida, porque o valor de retorno deve ser depositado em uma variável, e "`3*x`" não é uma variável.[1]

Em princípio qualquer tipo de dado pode ser passado como parâmetro, seja ele pré-definido (`integer`, `char`, etc) ou criado pelo programa (declaração `type`). A norma prevê até mesmo a passagem de procedimentos e funções como parâmetros de outros procedimentos ou funções, mas esta funcionalidade não é aceita por todos os compiladores, e não será objeto de estudo neste curso.

Finalmente, devemos lembrar que um procedimento implementa um sub-algoritmo. Desta forma ele deve ser visto como um bloco de programa com uma certa autonomia, e sua interface (parâmetros) deve ser bem projetada para ser genérica e facilitar o uso posterior do mesmo em outras situações.

5.1.3 Projetando um procedimento

Vamos projetar passo a passo um procedimento para ordenar vetores usando o método da bolha. Desejamos passar ao procedimento o vetor a ordenar, e recebe-lo de volta ordenado. O código o algoritmo da bolha foi apresentado na seção 4.1. Para projetá-lo vamos seguir os seguintes passos:

[1] Na realidade a diferença entre parâmetros normais e parâmetros `var` reside no fato de que os primeiros são passados *por valor* e os últimos *por referência*. Em outras palavras, na passagem de um parâmetro normal uma cópia do mesmo é entregue ao procedimento, que a descarta no final de sua execução; na passagem de um parâmetro `var` é entregue apenas uma referência (endereço) da variável indicada, e portanto as operações serão efetuadas sobre a própria variável, e não mais em uma cópia da mesma.

1. Vamos definir a interface do procedimento, ou seja, seu nome e os parâmetros que ele recebe e/ou devolve. Vamos usar o nome `Bolha`; ele recebe um vetor de reais `vet` e o seu número de elementos `tamanho`, e deve devolver o mesmo vetor, ordenado. Portanto `vet` é um parâmetro de entrada e saída, e `tamanho` apenas de entrada. Com isso a interface do nosso procedimento fica assim:

```
procedure Bolha (var Vet : VetorReal ;  tamanho : integer) ;
```

2. O corpo do procedimento implementa o algoritmo da bolha, como vimos anteriormente:

```
repeat
  HouveTroca := false ;
  for i := 1 to tamanho-1 do
    if Vet[i] > Vet[i+1] then
      begin
        Auxiliar := Vet[i] ;
        Vet[i]   := Vet[i+1] ;
        Vet[i+1] := Auxiliar ;
        HouveTroca := true ;
      end ;
until not HouveTroca ;
```

3. Para definir o contexto local do procedimento, devemos analisar seu corpo e determinar as variáveis, constantes, tipos e procedimentos necessários à implementação do mesmo. Algumas variáveis são recebidas como parâmetros, e portanto não precisam ser declaradas localmente. Observando o código acima, podemos construir o seguinte contexto, com as variáveis necessárias à implementação do algoritmo da bolha:

```
var
  i          : integer ;
  Auxiliar   : real ;
  HouveTroca : boolean ;
```

   Devemos observar que a declaração do tipo `VetorReal` é externa ao procedimento, pois esse tipo deve ser conhecido de quem ativa o procedimento (para poder passar o vetor como parâmetro).

Juntando os elementos acima temos a declaração completa do procedimento `Bolha`:

```
procedure Bolha (var Vet : VetorReal ;  tamanho : integer) ;
var
  i          : integer ;
  Auxiliar   : real ;
  HouveTroca : boolean ;

begin
  repeat
    HouveTroca := false ;
    for i := 1 to tamanho-1 do
      if Vet[i] > Vet[i+1] then
        begin
          Auxiliar := Vet[i] ;
          Vet[i]   := Vet[i+1] ;
          Vet[i+1] := Auxiliar ;
          HouveTroca := true ;
        end ;
  until not HouveTroca ;
end ;
```

O programa completo, usando o procedimento `Bolha`, pode ser visto a seguir. Observe como o programa principal ficou mais simples e claro:

```
program bolha ;
uses crt ;
type
  VetorReal = array [1..1000] of real ;

var
  V   : VetorReal ;
  i,n : integer ;

procedure Bolha (...) ;
...

begin { programa principal }
  readln (n) ;                      { leitura do vetor }
  for i := 1 to n do
    readln (V[i]) ;
  Bolha (V,n) ;                     { ordenar o vetor }
  writeln ('Vetor ordenado:') ;     { escrever resultado }
  for i := 1 to n do
    writeln (V[i]:10:4) ;
end.
```

5.2 Funções

Certos blocos de programa buscam determinar um resultado específico e único a partir dos dados de entrada que recebe, como por exemplo o determinante de uma equação do 2º grau a partir dos valores dos coeficientes de $f(x) = ax^2 + bx + c$. Neste caso podemos construir uma estrutura similar ao procedimento, mas que permite retornar esse valor de forma mais simples e clara: a *função*. Existem muitas funções pré-definidas em Pascal, como `sin(x)`, `abs(x)`, e

esta seção mostra como é possível criar outras. A declaração de uma nova função é similar à de um procedimento, com duas ressalvas: devemos indicar o tipo do resultado que ela retorna, e no bloco de instruções devemos atribuir um valor a esse resultado. A forma genérica de uma declaração de função é apresentada abaixo:

```
function nome da função ( parâmetros) : tipo do resultado ;
  declarações locais
begin
  instruções
  ...
  nome da função := ... ;
end;
```

Seguindo esta forma, nossa função para o cálculo do determinante ficaria assim:

```
function determ (a, b, c : real) : real ;
begin
  determ := sqr(b) - 4 * a * c ;
end ;
```

Esta função pode ser usada exatamente da mesma forma que as funções pré-definidas:

```
readln (c2, c1, c0) ;
if determ (c2, c1, c0) >= 0 then
  calcula raízes reais
else
  calcula raízes complexas ;
```

Vamos a um segundo exemplo, implementando uma função chamada `sinal(x)`, que retorna -1 se $x < 0$, 0 se $x = 0$ e $+1$ se $x > 0$:

```
function sinal (x : real) : integer ;
begin
if x < 0 then
  sinal := -1
else
  if x > 0 then
    sinal := 1
  else
    sinal := 0 ;
end ;
```

As funções são similares aos procedimentos no que diz respeito aos parâmetros de entrada ou entrada e saída, declarações locais e escopo de variáveis. Todavia, o valor de retorno de uma função deve pertencer a um tipo básico (`integer`, `byte`, `real`, `character`, `boolean`, etc), ou seja, não pode retornar dados de tipo `array` ou `record`. No entanto alguns compiladores recentes podem fazê-lo, e a norva norma prevista para a linguagem Pascal deve incorporar essa facilidade.

5.3 Escopo das variáveis

Vimos que é possível declarar variáveis no interior de procedimentos ou funções, e que estas só são reconhecidas dentro de certos limites. Esses limites são conhecidos como *escopo de uma variável*. Algumas regras permitem definir o escopo de uma variável:

- Uma variável só existe no módulo onde foi declarada, e deixa de existir com o fim da execução deste.
- Uma variável declarada em um módulo também é acessível a seus módulos internos. Da mesma forma, uma variável declarada no programa principal é acessível a todos os módulos do programa.
- Uma variável pode ser redefinida para um módulo (e seus módulos internos). Essa redefinição vale enquanto durar a execução do módulo; fora deste, volta a valer a definição anterior.

O exemplo a seguir permite compreender melhor essas regras:

```
program escopos ;
var
  J,K : integer ;

  procedure P1 ;
  var
    K : char ;

    procedure F1 ;
    var
      X,Y : real ;
    ...


  ...


  function F2 ;
  var
    X: string[10] ;
  ...


...
```

Podemos observar que a variável `J` é acessível a todos os módulos, enquanto `Y` só é acessível a `F1`. No interior de `P1` (e portanto de `F1`), a variável `K` definida em `P1` toma o lugar de `K` definida globalmente. As variáveis `X` definida em `F1` e `X` definida em `F2` são completamente independentes, e não se interferem.

5.4 Recursividade

Em alguns situações pode ser útil a um procedimento ou função chamar a si próprio durante sua execução. Por exemplo, o fatorial de um número inteiro $n > 0$, indicado por $n!$, é normalmente calculado assim:

$$n! = n \times (n-1) \times (n-2) \times (n-3) \times ... \times 1$$

Essa definição nos permite definir a seguinte implementação:

```
function fatorial (n : integer) : integer ;
var
  i, fat : integer ;
begin
  fat := 1 ;
  for i := 2 to n do
    fat := fat * i ;
  fatorial := fat ;
end ;
```

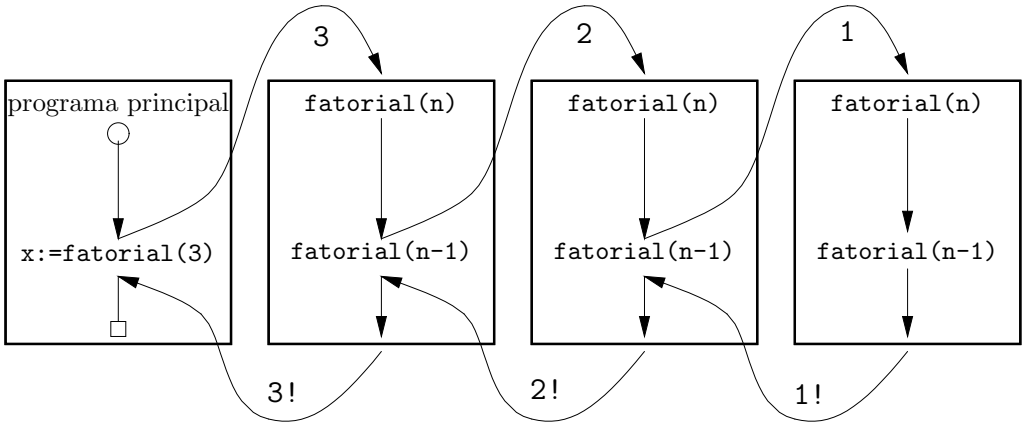
Matematicamente também podemos descrever o fatorial de um número desta forma:

$$n! = \begin{cases} 1 & \text{se } n \leq 1 \\ n \times (n-1)! & \text{se } n > 1 \end{cases}$$

Esta definição nos permite reescrever a função `fatorial(n)` da seguinte forma:

```
function fatorial (n : integer) : integer ;
begin
  if n > 1 then
    fatorial := n * fatorial (n-1)
  else
    fatorial := 1 ;
end ;
```

Quando executamos o comando `x := fatorial(3)`, o programa executa a seguinte seqüência de chamadas:



No exemplo acima a função `Fatorial` chama a si própria para ajudar no cálculo da solução. Chamamos isto *recursividade*, e dizemos que `fatorial` é uma função recursiva. A recursividade pode ser muito útil na resolução de problemas envolvendo estruturas de dados complexas. Assim como ocorre com as funções, podemos também escrever procedimentos recursivos.

5.5 Bibliotecas

Até agora os programas que construímos fazem uso de uma biblioteca chamada `crt`, através do comando `uses`. Uma biblioteca é na verdade um conjunto de declarações de procedimentos, funções, tipos, constantes e variáveis que servem a um determinado objetivo. No caso da biblioteca `crt`, ela oferece os serviços básicos de entrada e saída (comandos `read`, `write`, etc). Em Pascal uma biblioteca é denominada uma `unit`. Existem diversas bibliotecas oferecidas pelo sistema, como `crt`, `printer`, `graph`, `dos`, etc.

O programador também pode construir suas próprias bibliotecas, para organizar seus programas e facilitar a criação de novos programas. A declaração de uma biblioteca é independente de um programa principal, e deve ser colocada em um arquivo separado de qualquer outro código. Sua forma básica é a seguinte:

```
unit nome ;
uses
  outras bibliotecas usadas por esta biblioteca.

interface
  tipos, constantes, variáveis e interfaces de procedimentos e
  de funções acessíveis aos programas que usam a biblioteca.

implementation
  tipos, constantes, variáveis, procedimentos e funções
  acessíveis somente a módulos desta própria biblioteca.

  implementação (corpo) dos módulos declarados na seção interface

begin
  código de inicialização da biblioteca, executado
  automaticamente no início da execução.
end.
```

O estudo detalhado da construção de bibliotecas foge ao objetivo deste texto. O leitor interessado poderá obter informações detalhadas e exemplos em [O'B92].

5.6 Exercícios

1. Escreva um procedimento chamado `Bolha`, para ordenar vetores de reais em ordem crescente ou decrescente, usando o método da bolha. Devem ser passados como parâmetros o vetor a ordenar, o número de elementos do mesmo e um valor lógico indicando se a ordem final deve ser crescente ou decrescente.

2. Elabore uma função que verifique se um número dado é par, retornando `true` ou `false`.

3. Elabore uma função que retorne o reverso de um número inteiro positivo (por exemplo, `reverso(4532) = 2354`).

4. Construa um procedimento que receba três inteiros e os devolva em ordem crescente.

5. A série de Fibonacci, bastante útil em biologia, pode ser definida matematicamente por:

$$\mathcal{F}(n) = \begin{cases} 1 & \text{se } n \leq 2 \\ \mathcal{F}(n-1) + \mathcal{F}(n-2) & \text{se } n > 2 \end{cases}$$

Escreva uma função recursiva para calculá-la, e um programa que gere a série de Fibonacci até o termo de ordem 20 ($n = 20$).

6. Construa um programa Pascal que leia um vetor com dez fichas contendo cada uma as seguintes informações relativas a uma pessoa: nome, sexo, idade, peso e altura. Devem ser criados procedimentos para:

   - A leitura de uma ficha;
   - a escrita de uma ficha;
   - indicar as informações coincidentes entre duas fichas passadas como parâmetros;
   - a partir de um nome passado como parâmetro, apresentar as informações associadas a este.

7. Os demais exercícios do capítulo 7 do livro [GL85].

Capítulo 6

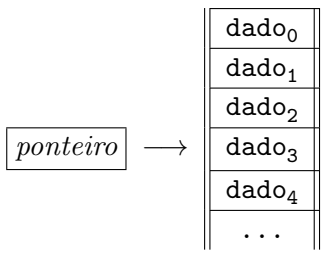
Manipulação de arquivos

As informações manipuladas no interior de um programa Pascal normalmente só existem durante sua execução. Uma vez encerrado o programa, todos os conteúdos das variáveis usadas são perdidos. Programas que usam uma grande quantidade de dados, como por exemplo a gestão de folhas de pagamento, seriam inviáveis sem formas de armazenamento auxiliares. Para contornar esse problema, podemos armazenar dados em arquivos nos discos do computador, para acessá-los quando necessários. Em Pascal existem dois tipos de arquivos: os arquivos de dados e os arquivos de texto.

6.1 Arquivos de dados

Um arquivo de dados permite armazenar uma grande quantidade de dados de um mesmo tipo, em seqüência. O tipo dos dados armazenados é definido no momento da criação dos dados, e pode ser um tipo básico (inteiro, real, caractere, etc) ou um registro.

A linguagem Pascal permite a manipulação de arquivos de dados em disco de forma versátil, embora bastante simples. Em Pascal, um arquivo de dados pode ser visto como um vetor de dados de um mesmo tipo armazenado no disco, com uma variável especial chamada *ponteiro do arquivo* que indica a próxima posição a acessar (ler ou escrever) no arquivo:



O acesso a um arquivo de dados em disco é efetuado através de uma variável especial chamada *descritor do arquivo*, que atua como intermediário entre o programa e o arquivo real em todas as operações envolvendo o mesmo. O descritor de arquivo armazena internamente o ponteiro de arquivo, que não pode ser acessado diretamente pelo programador. Vejamos a declaração de um descritor de arquivo de fichas de livros:

```
type
  FichaLivro = record
    Nome    : string [50] ;
    Autor   : string [30] ;
    Editora : string [15] ;
    Código  : string [10] ;
    Páginas : integer ;
    Emprest : boolean ;
  end ;

var
  ArqLivros : file of FichaLivro ;
```

A variável `ArqLivros` é o descritor de um arquivo de dados podendo conter um número qualquer de elementos do tipo `FichaLivro`, limitado somente pelo espaço disponível em disco. Para poder ser usado, esse descritor deve ser associado a um arquivo real no disco, através do comando `assign`:

```
assign (ArqLivros, 'c:livros.dat') ;
```

A partir dessa associação todas as operações efetuadas sobre o descritor de arquivo `ArqLivros` serão automaticamente refletidas sobre o arquivo `c:livros.dat`. Nosso próximo passo é abrir o arquivo, ou criá-lo se ele ainda não existe. Para abrir um arquivo já existente, usamos o comando `reset (ArqLivros)`. Se o arquivo associado ainda não existe (ou se existe mas desejamos apagar seu conteúdo anterior), usamos o comando `rewrite (ArqLivros)`.

Aqui surge um problema: para que possamos escolher corretamente entre `reset` e `rewrite`, precisamos saber se o arquivo desejado já existe no disco ou não. Se usarmos o comando `reset` para tentar abrir um arquivo não existente, um error será detectado, causando a interrupção do programa. A resposta a esse problema depende fortemente do compilador usado. Vamos apresentar a solução empregada no ambiente *Turbo-Pascal*: este ambiente emprega estruturas especiais chamadas *diretivas de compilação* para alterar o comportamento do compilador. Em nosso caso vamos usar a diretiva `$I`, que permite desativar temporariamente o controle de erros de entrada e saída. Para saber se ocorreram erros enquanto o controle estiver desativado, devemos consultar uma função especial chamada `IOresult`, que retorna o código numérico do último erro ocorrido (o código 0 (zero) corresponde à ausência de errros).[1]

A diretiva de compilação `$I` deve ser declarada entre chaves, com o sinal `-` para desativar o controle de erros e `+` para reativá-lo. Com isso podemos escrever o seguinte trecho de código para a abertura de um arquivo em disco:

```
assign (ArqLivros, 'c:livros.dat') ;  { associa descritor ao arquivo real  }
{$I-}                                 { desliga o controle de erros de E/S }
reset (ArqLivros) ;                   { tenta abrir o arquivo              }
{$I+}                                 { religa o controle de erros de E/S  }
if IOresult <> 0 then                 { se houve erro então                }
  rewrite (ArqLivros) ;               { cria o arquivo                     }
```

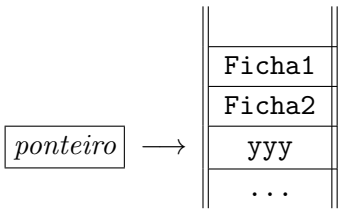
Agora podemos acessar o arquivo aberto para leituras e escritas. Para tal usaremos os já conhecidos comandos `read` e `write`, de uma forma um pouco diferente: devemos indicar como

[1] Para maiores informações sobre o controle de erros de E/S, as diretivas de compilação e a função `IOresult`, consulte o manual do compilador, ou [O'B92].

primeiro parâmetro o descritor de arquivo a acessar, e como segundo parâmetro a variável a ler ou escrever[2]. O acesso ao arquivo é efetuado de maneira seqüencial, ou seja: o ponteiro do arquivo recém-aberto indica o primeiro elemento do arquivo, e a cada operação de leitura ou de escrita ele avança automaticamente ao próximo elemento.

```
read  (ArqLivros, Ficha1) ;
write (ArqLivros, Ficha2) ;
```

O comando `read` lê o elemento na posição indicada pelo ponteiro do arquivo, colocando seu valor na variável indicada, e avança o ponteiro ao elemento seguinte. Uma tentativa de leitura após o último elemento do arquivo resulta em erro. O comando `write` escreve o conteúdo da variável indicada sobre a posição indicada pelo ponteiro do arquivo, e também o avança. Uma escrita após o último elemento do arquivo insere o novo valor no final do mesmo. Após a execução dos comandos acima teremos a seguinte situação no arquivo:



A função `eof(`*descritor)* (do inglês *end-of-file*) retorna `true` quando o ponteiro chegou ao final do arquivo. Vamos aos exemplos: o trecho de programa abaixo permite ler todos os elementos do arquivo de fichas de livros e colocá-los em um vetor de fichas na memória (consideramos que todas as variáveis mencionadas foram previamente declaradas):

```
assign (ArqLivros, 'c:livros.dat') ;
reset (ArqLivros) ;
NumFichas := 0 ;
while not eof (ArqLivros) do
  begin
    read (ArqLivros, UmaFicha) ;
    NumFichas := NumFichas + 1 ;
    Ficha [NumFichas] := UmaFicha ;
  end ;
close (ArqLivros) ;
```

No final do programa podemos transferir novamente todas as fichas do vetor de fichas para o arquivo. Como o vetor pode ter sido reduzido pela supressão de fichas, deve-se abrir o arquivo usando o comando `rewrite`, para apagar seu conteúdo anterior:

```
rewrite (ArqLivros) ;
for i := 1 to NumFichas do
  write (ArqLivros, Ficha[i]) ;
close (ArqLivros) ;
```

[2]Os dados são armazenados no arquivo em disco usando o formato binário de representação empregado internamente pelo compilador. Desta forma, um arquivo de dados não pode ser examinado ou alterado usando um editor externo.

Note que não foi necessário executar o comando `assign` novamente. Quando uma associação entre descritor e arquivo físico é efetuada, ela permanece válida em todo o escopo do descritor, durante toda a execução do programa.

Diversos comandos e funções são definidos em Pascal para o acesso a arquivos de dados. Abaixo estão indicados alguns exemplos ; uma lista mais ampla pode ser obtida nos manuais do compilador utilizado.

`filesize(`*descritor)* : função que retorna um valor inteiro indicando o número de elementos do arquivo associado ao descritor.

`filepos(`*descritor)* : função que retorna um valor inteiro indicando a posição atual do ponteiro do arquivo associado ao descritor (o primeiro elemento está na posição 0).

`seek(`*descritor, posição)* : procedimento que posiciona o ponteiro do arquivo na posição dada pelo valor inteiro *posição*. Este procedimento permite o acesso aleatório aos elementos do arquivo, ou seja, sem precisar respeitar a ordem seqüencial.

`erase(`*descritor)* : procedimento que apaga o arquivo associado ao descritor.

`rename(`*descritor, string)* : procedimento que permite mudar o nome do arquivo associado ao descritor para o nome contido em string.

6.2 Arquivos de texto

Quando usamos os comandos `read` e `readln` para ler dados de entrada do teclado, ou os comandos `write` e `writeln` para escrever mensagens na tela, na verdade estamos executando ações sobre arquivos de um tipo muito especial: os *arquivos de texto*. O compilador normalmente pré-define três arquivos de texto para as operações padrão de entrada e saída: o arquivo `Input`, normalmente associado ao teclado, o `Output`, associado à tela e o `Lst`, associado à impressora. A declaração desses arquivos e sua associação aos dispositivos físicos são implícitas, o que torna os comandos a seguir totalmente equivalentes:

```
read (NumAlunos) ;
writeln ('O número de alunos é ', NumAlunos:3) ;

 ou

read (Input, NumAlunos) ;
writeln (Output, 'O número de alunos é ', NumAlunos:3) ;
```

Os arquivos de texto são estruturados como seqüências de caracteres agrupados em linhas, e podem ser acessados através dos procedimentos padrão de entrada e saída `read`, `readln`, `write` e `writeln`. Além dos arquivos pré-definidos, o programador pode criar outros, usando o tipo especial `text`, e associá-los a arquivos em disco, que podem então ser acessados da mesma forma que o teclado ou a tela. Vejamos o exemplo a seguir:

```
program arquivo_texto ;
uses crt ;
var
   Relatorio : text ;
   NumAlunos : integer ;
begin
   NumAlunos := 21 ;
   assign  (Relatorio, 'c:relat.txt') ;
   rewrite (Relatorio) ;
   writeln (Relatorio, 'O número de alunos é ', NumAlunos:3) ;
   close   (Relatorio) ;
end.
```

Após a execução do programa acima teremos criado um arquivo em disco chamado `relat.txt` contendo exatamente o que obteríamos caso a escrita fosse feita na tela:

```
O número de alunos é  21
```

Alguns cuidados especiais devem ser observados na manipulação de arquivos de texto:

- Um arquivo de texto só pode ser acessado em seqüência, ou seja, operadores como `seek` e `filepos` não podem ser usados sobre esse tipo de arquivo.
- Normalmente um arquivo de texto deve ser aberto para somente para leitura ou somente para escrita, mas nunca para os dois simultaneamente. No entanto nada impede que um arquivo seja aberto diversas vezes em um mesmo programa, algumas delas para leitura e outras para escrita.

Uma das propriedades mais interessantes dos arquivos de texto é a conversão automática dos dados lidos ou escritos. Da mesma maneira como ocorre na escrita em tela, os dados numéricos escritos em um arquivo de texto são automaticamente convertidos para uma seqüência de caracteres, que também pode ser formatada. O mesmo ocorre no processo de leitura, no qual os dados numéricos em forma de caracteres são convertidos para sua representação interna em binário.

Além da função de teste de final de arquivo `eof(`*descritor*`)`, pode ser usada nos arquivos de texto uma função de teste de final de linha `eoln(`*descritor*`)`, que retorna `true` caso o ponteiro do arquivo se encontre no final de uma linha.

6.3 Exercícios

1. Escreva um programa Pascal para manter um cadastro de fichas de alunos em disco. Cada ficha deve conter o nome, idade e média final do aluno. O programa deve ter procedimentos para:
   - Ler dados de um aluno do teclado e colocar em uma ficha.
   - Acrescentar uma ficha ao cadastro.
   - Remover uma ficha do cadastro.
   - Escrever os dados de uma ficha na tela.
   - Ler todas as fichas que estão no disco.

- Guardar todas as fichas no disco.

2. Acrescentar ao programa do exercício anterior procedimentos para:

   - Produzir um arquivo de listagem em disco, no qual cada linha contém o nome do aluno e sua idade (cuidar da formatação).
   - Ler uma ficha de aluno contida em um arquivo em disco, escrita no seguinte formato:

     ```
     Nome do aluno
     idade   média
     Nome do aluno
     idade   média
     ...
     ```

3. Usando a diretiva de compilação `$I`, escrever uma função para testar se um arquivo existe no disco ou não. A função `ExisteArquivo` deve receber como parâmetro de entrada o nome do arquivo e retornar um valor lógico indicando a existência do arquivo.

4. Escreva um programa para extrair as palavras contidas em um arquivo de texto. Palavras são seqüências de dois ou mais caracteres, separadas por um ou mais espaços em branco, dígitos, pontuação, etc. O programa deve produzir um arquivo texto contendo todas as palavras encontradas em ordem crescente, e indicando quantas vezes cada palavra ocorreu no texto lido.

5. Escreva um programa que leia dois arquivos texto com nomes de pessoas, ordenados em ordem alfabética crescente, e indique na tela os nomes comuns aos dois arquivos. Não devem ser usados vetores para armazenar os nomes na memória; os testes, comparações e a escrita da saída devem ser efetuados na medida em que os nomes são lidos dos dois arquivos.

Capítulo 7

Tipos enumerados e conjuntos

Neste capítulo veremos alguns tipos de dados pré-definidos em Pascal que permitem uma maior versatilidade e clareza na construção de programas: os *tipos enumerados*, as *sub-faixas* e os *conjuntos*.

7.1 Tipos enumerados

Geralmente é possível expressar os dados que desejamos utilizando os tipos básicos do Pascal (`integer`, `byte`, `real`, etc), ou tipos compostos a partir desses tipos básicos (`record`, `array`, etc). Por exemplo, se desejarmos definir uma variável para guardar o naipe de uma carta em um programa para jogar truco, podemos escrever `var naipe : byte ;` e utilizar alguma forma de codificação para representar os naipes, por exemplo: 1=ouro, 2=espada, 3=copas e 4=paus. Podemos agir da mesma forma para armazenar os meses do ano, os dias da semana, as estações do ano, os estados de um semáforo, etc.

Todavia o uso desse tipo de codificação numérica pode tornar mais obscuros os programas, e assim favorecer a ocorrência de erros. A linguagem Pascal oferece uma ferramenta poderosa para a definição de tipos de variáveis adequados a situações como essas: os *tipos enumerados*. Eles são chamados assim porque em sua definição são enumerados os valores possíveis para as variáveis desse tipo (normalmente são possíveis até 256 valores distintos para cada tipo). Vejamos como ficaria a definição de alguns tipos enumerados:

```
type
  TipoNaipe   = (ouro, espada, copas, paus);
  TipoDia     = (segunda, terça, quarta, quinta,
                 sexta, sábado, domingo) ;
  TipoEstação = (outono, inverno, primavera, verão) ;
  TipoSinal   = (apagado, verde, amarelo, vermelho) ;
  TipoFruta   = (banana, maçã, pera, uva, limão, manga, abacaxi, mamão,
                 abacate, laranja, melancia, pitanga, ameixa) ;
```

Após sua definição esses tipos podem então ser usados normalmente na declaração de variáveis:

```
var
  Naipe : TipoNaipe ;
  Sinal : TipoSinal ;
```

Durante o programa, a variável `Naipe` declarada acima só pode assumir um dos quatro valores enumerados na definição do tipo `TipoNaipe`, ou seja, `ouro`, `espada`, `copas` ou `paus`. Esses valores também podem ser usados durante o programa. O mesmo ocorre para as outras variáveis:

```
...
if (Naipe = copas) or (Naipe = ouro) then
  writeln ('Carta vermelha !') ;
...
case Sinal of
  apagado  : writeln ('Avance, pois o sinal está pifado.') ;
  verde    : writeln ('Pode passar tranqüilo') ;
  amarelo  : writeln ('Pode ir que dá tempo !') ;
  vermelho : if NumGuardas = 0 then
               writeln ('Vai, que não tem nenhum guarda.')
             else
               writeln ('Espere o guarda sair...') ;
end ;
```

Estabelecemos a ordem de precedência dos valores possíveis para um tipo enumerado em dua definição. Essa ordem é fixa e pode ser usada durante o programa. Por exemplo, no tipo `TipoNaipe` a relação entre os valores enumerados é `ouro` $<$ `espada` $<$ `copas` $<$ `paus`. Assim, podemos escrever:

```
{ Minha e Sua são dois registros contendo um valor e um naipe }
if (Minha.Valor = Sua.Valor) then
  if Minha.Naipe > Sua.Naipe then
    writeln ('Ganhei !')
  else
    writeln ('Essa era só treino...') ;

if (Sinal < vermelho) or (NumGuardas = 0) then
  writeln ('pode passar...')
else
  writeln ('espere o guarda sair...') ;
```

Essa relação de ordem entre os valores possíveis em um tipo enumerado permite seu uso de maneira bastante interessante. Por exemplo, podemos escrever um laço *para-faça* para varrer os naipes possíveis:

```
for Naipe:= ouro to paus do
  begin
    ...
  end ;
```

Podemos também usar um tipo enumerado na definição e manipulação de arranjos:

```
type
  Meses = (janeiro,fevereiro,março,abril,maio,junho,julho,
           agosto,setembro,outubro,novembro,dezembro);
var
  NumDias : array [Meses] of byte ;
  Mes     : Meses ;
  Ano     : integer ;

begin
  NumDias[janeiro] := 31 ;
  if Ano mod 4 = 0 then
    NumDias[fevereiro] := 29
  else
    NumDias[fevereiro] := 28 ;
  ...
  for Mes := janeiro to dezembro do
    writeln (NumDias[Mes]) ;
```

Atenção: os valores possíveis para os tipos enumerados só são reconhecidos no interior do programa. Eles não podem ser diretamente lidos do teclado ou escritos na tela[1]. Para ler ou escrever o valor de uma variável de tipo enumerado normalmente é necessário usar uma estrutura como a que segue:

```
case Naipe of
  paus  : write('paus');
  copas : write('copas');
  espada: write('espada');
  ouro  : write('ouro');
end;
```

Algumas funções são especialmente definidas para tratar tipos enumerados em Pascal:

`ord(x)` : devolve a posição de `x` na lista de valores possíveis, começando por 0 (zero). Assim, `ord(ouro)` devolve 0 e `ord(paus)` devolve 3.

`pred(x)` : devolve o valor anterior a `x`. Por exemplo, `pred(paus)` retorna o valor `copas`, e `pred(ouro)` resulta em erro.

`succ(x)` : devolve o valor posterior a `x`. Assim, `succ(ouro)` retorna o valor `espada`, e `succ(paus)` resulta em erro.

7.2 Sub-faixas

Os tipos pré-definidos `byte` e `integer` podem variar de capacidade em função do compilador usado. Para contornar esse problema e tornar os programas mais portáveis, muitas vezes é preferível declarar as variáveis em função dos valores que elas podem assumir. Isso é feito através da declaração de tipos *sub-faixas*. Podemos declarar sub-faixas de tipos inteiros, enumerados ou de caracteres, como mostra o exemplo a seguir:

[1] Alguns compiladores recentes podem faze-lo.

```
type
  Inteiro   = 1..1000 ;

var
  i,j : Inteiro ;
```

No exemplo, as variáveis `i` e `j` são de um tipo inteiro podendo variar entre 1 e 1000. O compilador ajustará automaticamente o espaço necessário para armazenar essas variáveis, otimizando a ocupação da memória e tornando o programa independente de detalhes internos do sistema.

Já empregamos sub-faixas de inteiros anteriormente, sem o perceber. A declaração de uma matriz, como vimos na seção 4.2, emprega sub-faixas de inteiros. Portanto, as duas declarações abaixo são equivalentes:

```
type
  TipoVetor1 = array [1..1000] of real ;
  TipoVetor2 = array [Inteiro] of real ;
```

Com o uso de diretivas adequadas, o compilador pode gerar automaticamente testes para verificar se as variáveis e índices se mantém dentro das faixas de valores para as quais foram definidas. No ambiente Turbo-Pascal isso pode ser obtido através da diretiva `$R`.

7.3 Conjuntos

Em Pascal é possível representar e tratar facilmente estruturas matemáticas de tipo conjunto. Os elementos de base de um conjunto devem ser dos tipos `char`, `byte` ou de tipos enumerados. Um conjunto pode conter um número qualquer de elementos do tipo básico usado, e não existindo uma ordem específica entre os elementos. Além disso, cada elemento pode apenas pertencer ao conjunto ou não (ou seja, um conjunto não pode conter dois elementos iguais). A declaração de um conjunto é bastante simples. Por exemplo, para definir um tipo conjunto cujos elementos serão caracteres, basta declarar:

```
type
  ConjCaracteres = set of char ;

var
  Letras, Digitos,
  Maiusculas, Minusculas,
  Vogais, Consoantes      : ConjCaracteres ;
```

As variáveis acima definidas são conjuntos podendo conter caracteres. As operações que podemos efetuar sobre um conjunto, ou entre conjuntos, são as seguintes (todas elas consideram que os conjuntos envolvidos são de mesmo tipo):

Atribuição : podemos atribuir um valor a um conjunto, indicando quais elementos este deverá conter. Por exemplo, podemos atribuir valores aos conjuntos `Letras` e `Dígitos`:

```
Pares    := ['0','2','4','6','8'] ;
Testados := [] ; { o conjunto vazio }
Digitos  := ['0'..'9'];
Letras   := ['A'..'Z','a'..'z'] ;
```

União : o conjunto união de dois ou mais conjuntos contém todos os elementos que percentem a um ou mais dos conjuntos envolvidos. Em Pascal a união de conjuntos é expressa pela operação '+':

```
Letras := ['A'..'Z'] + ['a'..'z'] ;
```

Podemos utilizar a operação de união para inserir um novo elemento em um conjunto: basta unir o conjunto dado a um conjunto contendo somente o elemento a inserir (no exemplo abaixo a variável `NovaLetra` é do tipo `char`):

```
readln (Novaletra) ;
Letras := Letras + [NovaLetra] ;
```

Intersecção : a intersecão entre dois ou mais conjuntos resulta em um conjunto cujos elementos pertencem a todos os conjuntos envolvidos. Em Pascal a intersecção entre conjuntos se efetua através do operador '*'. Por exemplo, o comando abaixo permite obter a intersecção entre o conjunto de vogais e o conjunto de mínúsculas, o que nos dá o conjunto de vogais minúsculas:

```
VogaisMin := Vogais * Minúsculas ;
```

Diferença : A diferença entre dois conjuntos `A` e `B` é um conjunto cujos elementos pertencem a `A` e não pertencem a `B`. Expressamos a diferença entre dois conjuntos em Pascal através do operador '-'. Assim, podemos obter os conjuntos de consoantes e de dígitos pares através dos seguintes comandos:

```
Consoantes := Letras - Vogais ;
Impares    := Dígitos - Pares ;
```

Além das operações acima descritas, podemos efetuar testes sobre variáveis de tipo conjunto. Os testes possíveis são descritos abaixo, e todos retornam valores lógicos (`true` ou `false`).

- $=$ (igual): testa a igualdade entre dois conjuntos: `if A=B then ...`
- $\neq$ (diferente): testa a desigualdade entre dois conjuntos: `if A<>B then ...`
- $\supset$ (contém): testa se o conjunto A contém o conjunto B: `if A>=B then ...`
- $\subset$ (está contido): testa se o conjunto A está contido no conjunto B: `if A<=B then ...`
- $\in$ (pertence): testa se um elemento pertence a um conjunto. Este operador é simbolizado pela palavra reservada `in`:

```
readln(NovaLetra) ;
if NovaLetra in Vogais then
  writeln (NovaLetra,' é uma vogal.')
else
  if NovaLetra in Consoantes then
    writeln (NovaLetra,' é uma consoante.')
  else
    if NovaLetra in Dígitos then
      writeln (NovaLetra,' é um dígito.')
    else
      writeln ('Não conheço o caratere ', NovaLetra);
```

Atenção: variáveis de tipo conjunto não podem ser diretamente lidas do teclado ou escritas na tela. Para ler um conjunto deve-se ler cada elemento separadamente e incluí-lo ao conjunto usando a operação de união. Para escrever um conjunto, deve-se percorre-lo testando se cada elemento pertence ou não ao conjunto, como mostra o exemplo abaixo:

```
program escreve_conjunto ;
type
  ConjCar = set of char ;

var
  Letras : ConjCar ;
  Letra  : char ;

begin
  Letras := ['A'..'K','W'] ;
  for Letra := 'A' to 'Z' do
    if Letra in Letras then
      write (Letra) ;
end.
```

7.4 Exercícios

1. Voce deve construir um programa que permita acompanhar o movimento de venda de cerveja em um bar, e ao final apresente a quantidade vendida para cada marca. O programa deverá ler os dados do teclado na forma "marca,quantidade". Para as marcas de cerveja devem ser usados tipos enumerados.

2. Escrever um programa para ler um texto caractere por caractere e mostrar quais letras ou dígitos apareceram no texto, usando conjuntos. Apresentar também o número de letras maiúsculas encontradas no texto.

3. Escrever um programa que leia um arquivo contendo um programa Pascal e determine quantas palavras reservadas indicando estruturas de controle foram encontradas. As palavras a encontrar são `for`, `while`, `repeat` , `if` e `case`. Para a contagem deve ser usado um vetor indexado por um tipo enumerado.

4. Escrever um procedimento que determine quantos elementos existem em um conjunto de letras passado como parâmetro.

5. Escrever um procedimento que escreva o conteúdo de um conjunto de letras maiúsculas. A saída deve seguir o seguinte modelo: `[A, C-J, M, N, T-Z]`.

Capítulo 8

Apontadores e estruturas dinâmicas

Em um programa Pascal, uma variável indica uma posição na memória do computador onde está armazenado um determinado dado. Quando manipulamos uma variável, na verdade estamos manipulando o conteúdo dessa posição de memória. A linguagem Pascal permite a manipulação dos endereços das variáveis através de um tipo especial de variável: os *apontadores*. O uso de apontadores combinados a registros permite a construção de estruturas de dados complexas como listas, pilhas, árvores e grafos, como veremos neste capítulo.

8.1 Apontadores

Por definição, um apontador é um tipo especial de variável que pode conter o endereço de outra variável. Em outras palavras, dizemos que um apontador aponta para outra variável, sendo desta forma uma referência para esta variável, que permite sua manipulação. Um apontador possui um tipo base, e só pode apontar para variáveis desse tipo. A declaração de um apontador usa o símbolo "^" antes do tipo base da variável apontada:

```
var
  nome : ^ tipo-base ;
```

Por exemplo, para criar apontadores para variáveis de tipo `real` e de tipo `char` escrevemos:

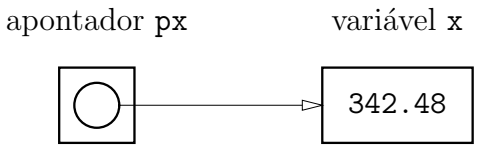
```
var
  ApReal : ^real ;
  ApChar : ^char ;
```

Neste caso a variável `ApReal` é um apontador que pode apontar para outras variáveis de tipo `real`. Para designar o conteúdo da posição de memória apontada pelo apontador `ApReal`, basta escrever `ApReal^`. Observe que por si só o apontador `ApReal` não contém nenhum valor real: ele só faz apontar para outras variáveis que contém reais. Veja o exemplo abaixo (a função `addr(x)` devolve o endereço na memória da variável `x` passada como parâmetro):

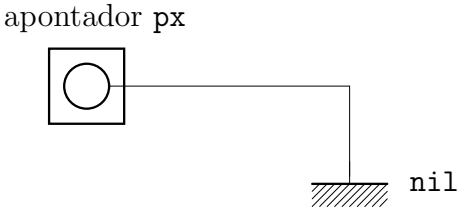
```
program Apontadores ;
uses crt ;
var
  x   : real ;
  apx : ^real ;

begin
  x   := 3.142 ;   { 1: atribui um valor a x }
  apx := addr(x) ; { 2: atribui a apx o endereço de x }
  writeln (apx^) ; { 3: vai escrever 3.142 }
  x := -2.718 ;    { 4: atribui um valor a x }
  writeln (apx^) ; { 5: vai escrever -2.718 }
  apx^ := 54.017 ; { 6: atribui indiretamente um valor a x }
  writeln (x) ;    { 7: vai escrever 54.017 }
end.
```

Na linha 2 o apontador `apx` recebe como valor o endereço de `x`, ou seja, ele passa a apontar para `x`. Assim a linha 3 vai escrever o valor contido em `x`. Mudando o valor de `x` (linha 4), o valor indicado por `apx^` também muda (linha 5). Podemos efetuar o caminho inverso, mudando o valor indicado por `apx^` (linha 6), o que afeta `x` (linha 7).

Como as estruturas de dados que os apontadores permitem construir podem ser bastante complexas, é comum representá-las graficamente:



O valor `nil` pode ser atribuido a um apontador, para indicar que ele não está apontando para lugar nenhum (atenção: um apontador não inicializado, ao qual nenhum valor foi atribuído, contém um valor qualquer, não necessariamente `nil`). Um apontador com valor `nil` é representado graficamente como um apontador aterrado:



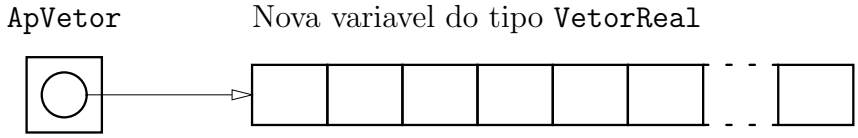
8.2 Alocação dinâmica de memória

A maior utilidade dos apontadores está na possibilidade de alocação dinâmica de memória. Quando uma variável é definida em um programa Pascal, um espaço na memória é reservado para armazená-la durante toda a execução do programa, e por isso essas variáveis são chamadas *estáticas*. O espaço disponível para guardar variáveis estáticas é relativamente pequeno (em geral 64 Kbytes nos PCs), bem menos que a memória total disponível na máquina, da ordem de dezenas de Mbytes. Durante a execução é possível usar essa memória extra para criar e destruir novas variáveis na medida em que forem necessárias. A isto chamamos *alocação dinâmica de memória*.

Para criar uma variável dinâmicamente, precisamos primeiro de um apontador que possa apontar para ela. Esse apontador será nossa única referência para a variável, e portanto deve ser manipulado com cuidado para que a variável não seja perdida. Digamos que desejamos criar dinâmicamente um vetor com 1000 elementos de tipo `real`. Primeiro precisamos definir seu tipo e criar um apontador para variáveis deste tipo:

```
type
  VetorReal = array [1..1000] of real ;

var
  ApVetor : ^VetorReal ;
```

Quando desejarmos criar uma variável dinâmica do tipo `VetorReal`, basta usar o comando `new(ApVetor)`. Ao executar essa operação, uma área de memória é reservada para alocar a variável, e o apontador `ApVetor` passa a apontar para ela:



O acesso ao vetor assim criado só é possível através do apontador que o referencia. Por exemplo:

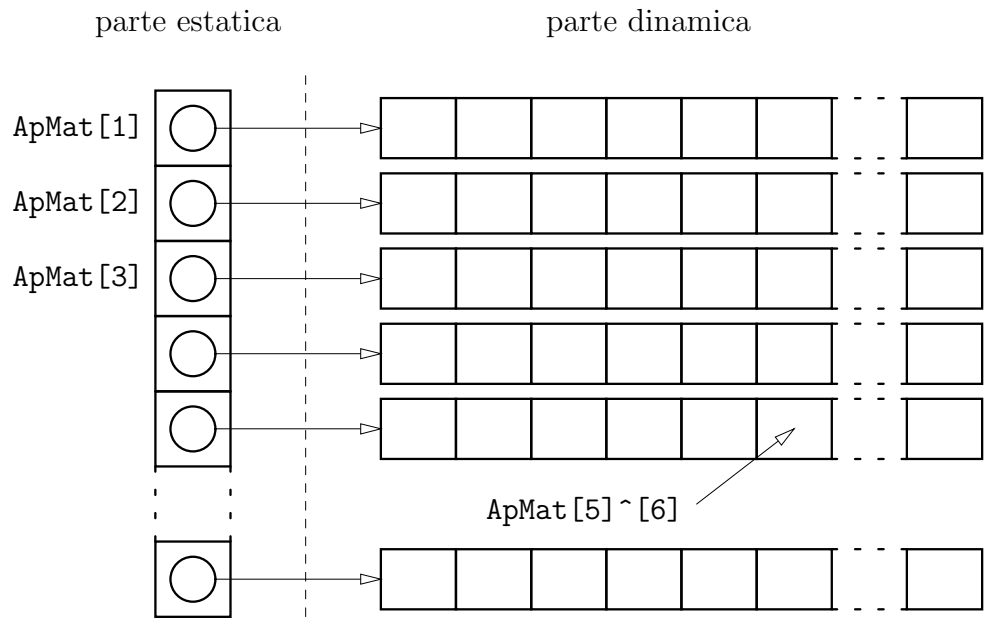
```
for i := 1 to 1000 do
  ApVetor^[i] := 0.0 ;
```

Quando uma variável dinâmica não é mais necessária, pode-se liberar a área de memória que ela ocupa. Para isso usa-se o operador `dispose(ApVetor)`, que recebe como parâmetro o apontador indicando a área a ser liberada.

Com o uso da alocação dinâmica de memória é possível construir estruturas de dados que seriam impossíveis de criar usando variáveis estáticas. Por exemplo, podemos criar uma matriz de 500 linhas e 1000 colunas contendo reais (uma matriz podendo conter 500.000 valores reais !). Para isso basta criar um vetor com 500 apontadores de vetores de reais, e depois alocar os 500 vetores. Ao final da execução a área dos 500 vetores deve ser liberada:

```
program alocacao ;
uses crt ;

type
  VetorReal = array [1..1000] of real ;  { definicao do tipo vetor       }

var
  ApMat : array [1..500] of ^VetorReal ; { vetor c/ 500 apontadores de vetores }
  i     : integer ;

begin
  for i := 1 to 500 do        { criar 500 vetores de reais dinamicamante }
    New (ApMat[i]) ;          { com alocacao da area em memoria          }
  ...
  ApMat[17]^[563] := 34.12 ;  { acesso a um elemento qualquer            }
  ...
  for i := 1 to 500 do        { delecao dos vetores, com liberacao       }
    Dispose (ApMat[i]) ;      { da area de memoria ocupada por eles      }
end.
```

Veja como ficaria graficamente essa estrutura:



8.3 Estruturas de dados dinâmicas

A principal utilidade dos apontadores e da alocação dinâmica de memória reside na construção de estruturas de dados complexas, que seriam de difícil construção usando variáveis estáticas. Esta seção apresenta algumas dessas estruturas, sem entretanto ter a intenção de se aprofundar no assunto. Os leitores interessados podem consultar [VSAF86, Sed83] para maiores detalhes e exemplos.

Os elementos básicos para a construção de estruturas de dados dinâmicas são o registro e o apontador. Uma estrutura dinâmica é geralmente composta por elementos de base chamados *nós*, que se referenciam entre si através de apontadores, permitindo assim construir estruturas

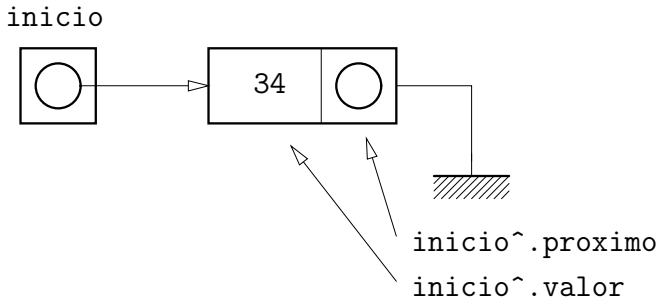
complexas como listas (simples ou duplas), pilhas, árvores, grafos, etc. Um nó é constituido por um registro contendo variáveis locais e apontadores para outros elementos do mesmo tipo (ou seja, outros nós):

```
type
  ApontaNo = ^No ;
  No = record
    valor   : integer ;
    proximo : ApontaNo ;
  end ;

var
  inicio : ApontaNo ;
```

O registro acima contém um valor qualquer e um apontador para uma variável do mesmo tipo. Esse registro permite a criação de uma lista de nós, com o início (primeiro elemento) sendo apontado pela variável `inicio`. Vamos ver agora como construir, passo a passo, uma lista com três nós. Vamos criar o nó inicial, e guardar nele o valor 34:

```
new(inicio) ;
inicio^.valor := 34 ;
inicio^.proximo := nil ; { sempre inicialize os apontadores }
```

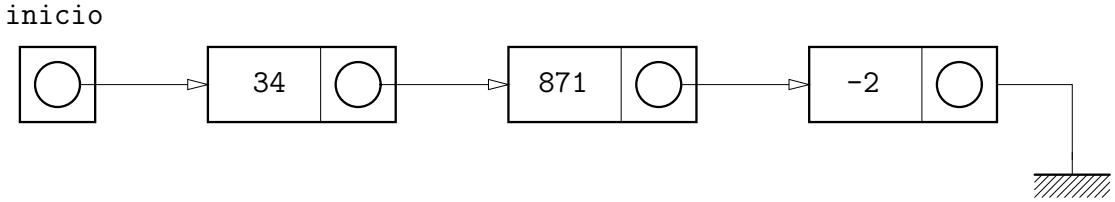
Com estes comandos teremos na memória uma estrutura como esta:



Vamos agora aumentar esta lista, inserindo mais dois elementos em seu final. Lembre-se de que a única referência que temos para a lista é o apontador `inicio`, e portanto todas as operações terão de ser feitas através dele:

```
new(inicio^.proximo) ;                      { cria novo nó }
inicio^.proximo^.valor := 871 ;
inicio^.proximo^.proximo := nil ;

new(inicio^.proximo^.proximo) ;             { cria novo nó }
inicio^.proximo^.proximo^.valor := -2 ;
inicio^.proximo^.proximo^.proximo := nil ;
```

Com estes comandos teremos na memória a seguinte estrutura:

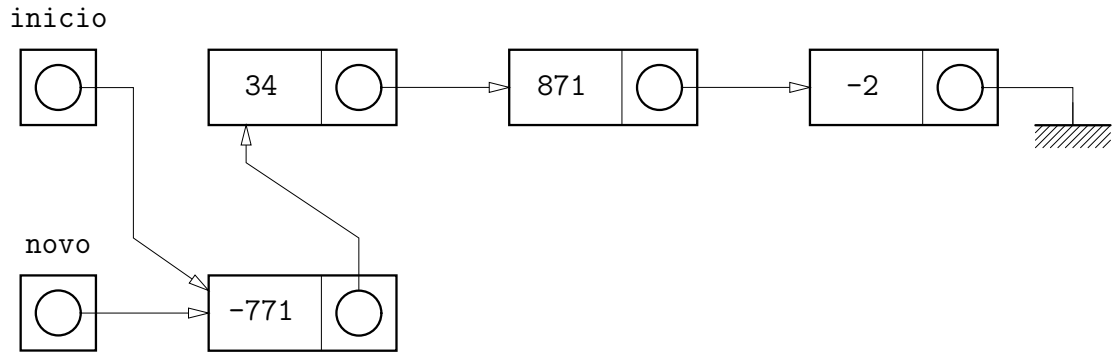


Essas operações nos permitiram criar uma série de nós, ligada pelos apontadores `proximo`, e cujo início é indicado pelo apontador `inicio`. Essa estrutura é normalmente chamada *lista encadeada simples*, e permite armazenar uma seqüência de valores, da mesma forma que um vetor. Mas, ao contrário deste, a lista só usa a memória efetivamente necessária para armazenar os valores utilizados, e pode ajustar seu tamanho em função da necessidade, até o limite da memória disponível no computador. Além disso, inserções e retiradas de elementos dessa lista exigem poucas operações. Por exemplo, a inserção de um novo nó na lista encadeada acima exige somente algumas operações com apontadores, pouco importa o tamanho da lista.

Como exemplo de manipulação de estruturas dinâmicas, vejamos o processo de inserção de um novo nó na lista acima criada. Dado um nó isolado, referenciado pelo apontador `novo`, vamos inseri-lo no início da lista apontada pelo apontador `inicio`. As operações necessárias a essa inserção são:

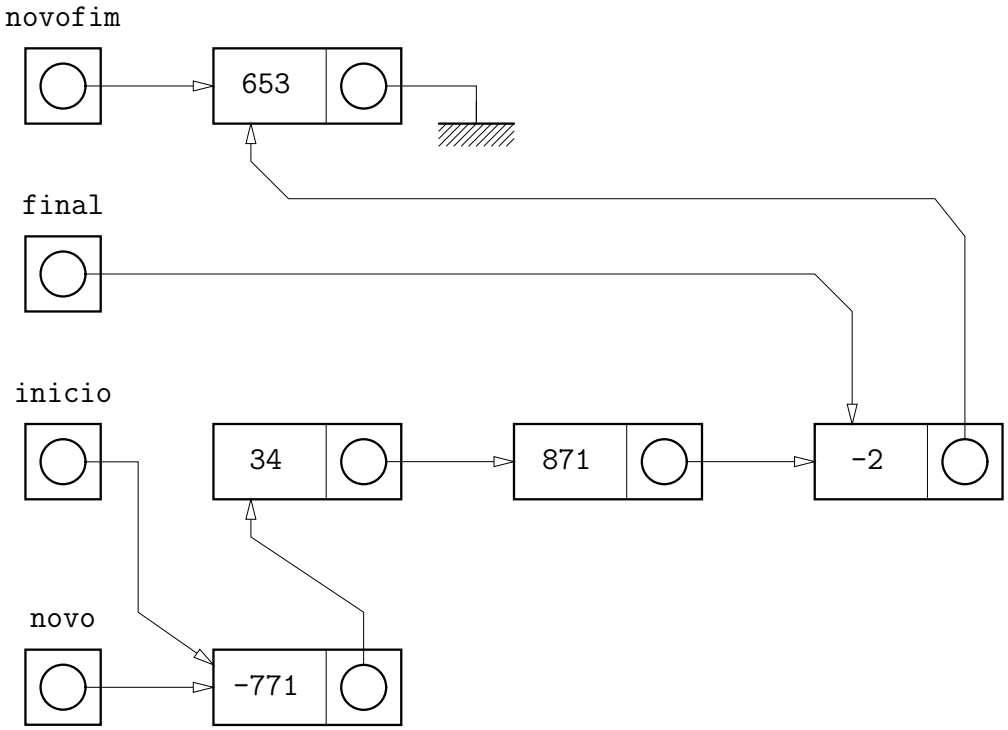
```
if ( inicio = nil ) then       { se início não apontar para nada então    }
  inicio = novo                { deve apontar para o mesmo local que novo }
else
  begin
    novo^.proximo := inicio ;  { lista apontada por início sucede novo    }
    inicio := novo ;           { novo passa a ser o nó inicial            }
  end ;
```

Apenas duas operações com apontadores foram necessárias (imagine o número de operações necessárias para inserir um novo valor no início de um vetor com milhares de elementos!). Após a inserção nossa lista terá a seguinte estrutura:



Vejamos agora ver como inserir um novo nó no final da lista. O nó a inserir está referenciado pelo apontador `novofim`. Antes de efetuar as manipulações de apontadores para inserir o nó na lista, devemos descobrir onde está o final da mesma. Partindo do início da lista, vamos avançar um apontador chamado `final` até que ele aponte para o último nó:

```
if ( inicio = nil ) then              { se início não apontar para nada então    }
  inicio = novofim                    { deve apontar para o mesmo local que novo }
else
  begin

    final := inicio ;                 { vamos avancar o apontador final até que  }
    while (final^.proximo <> nil) do  { ele aponte para o último nó da lista     }
      final := final^.proximo ;

    final^.proximo := novofim ;       { ultimo nó aponta para o novo nó          }
    novofim^.proximo := nil ;         { novo nó não aponta para mais nada        }
  end ;
```



É muito freqüente o uso de procedimentos e funções recursivas para percorrer ou manipular estruturas dinâmicas, devido certamente ao próprio caractere recursivo da definição dessas estruturas. Por exemplo, podemos escrever o seguinte procedimento para remover da memória uma lista encadeada simples como as anteriormente apresentadas:

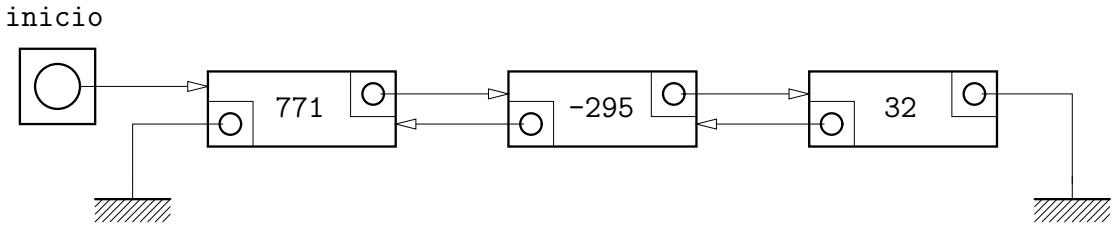
```
procedure ApagaLista (var Comeco: ApontaNo) ;
begin
  if Comeco <> nil then
    begin
      ApagaLista (Comeco^.proximo) ; { tenta avancar mais na lista        }
      Dispose (Comeco) ;             { apaga nó atual                     }
      Comeco := nil ;                { Dispose nao o faz automaticamente! }
    end ;
end ;
```

O funcionamento do código acima é bastante simples: o procedimento chama a si próprio recursivamente, passando como parâmetro um apontador para o restante da lista

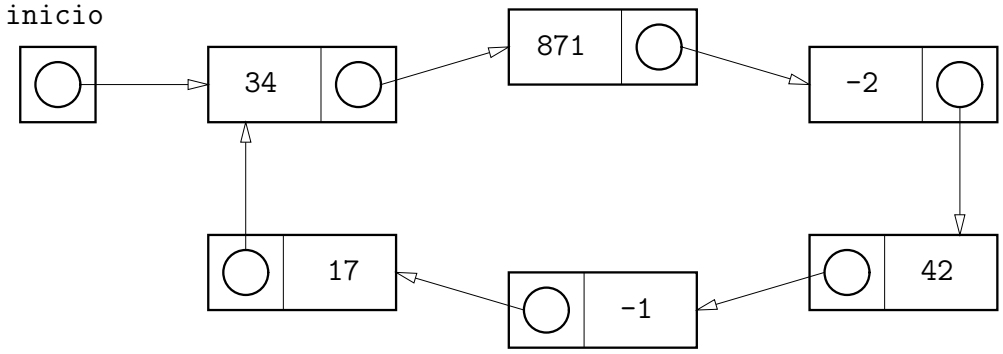
(`Comeco^.proximo`) até encontrar o final da lista (`Comeco = nil`). A partir daí ele vai voltando na lista (retornando das chamadas recursivas) e descartando os nós percorridos, um por um. Desta forma, para apagar completamente nossa lista encadeada bastaria:

```
ApagaLista (inicio) ;
```

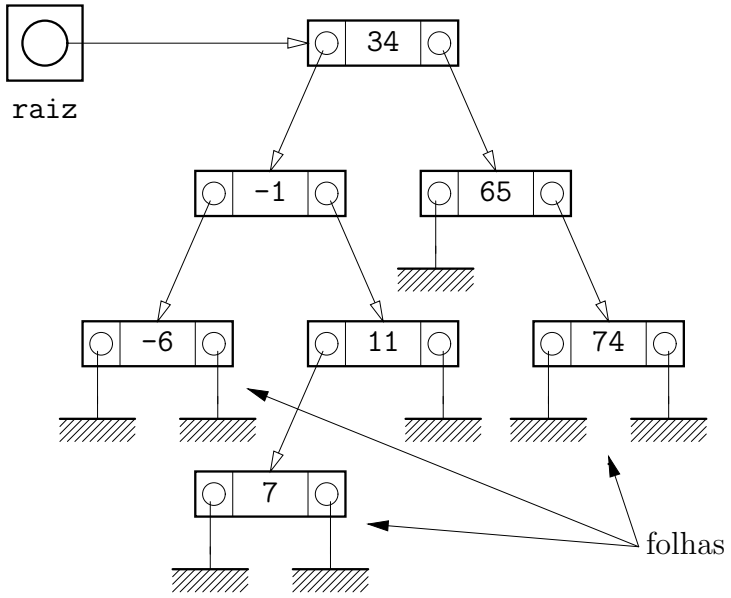
Vejamos agora algumas estruturas mais complexas: se cada elemento da lista possuir dois apontadores, um indicando o próximo elemento e outro indicando o elemento anterior, teremos uma *lista duplamente encadeada*:



Esta estrutura pode ser muito útil se precisarmos percorrer a lista em ambos os sentidos. Podemos também construir uma *lista circular*, simples ou dupla, quando o último elemento aponta de volta para o primeiro (e vice-versa, no caso de listas circulares duplamente encadeadas). A figura a seguir mostra uma lista circular simples:



Outro tipo de estrutura de dados dinâmica bastante empregada é a *árvore*. Nela, cada nó possui apontadores para *nós filhos*, e cada filho possui somente um nó anterior (seu *pai*). O nó principal da árvore, que não possui pai, é chamado *raiz*; um nó terminal (sem filhos) é chamado *folha*. Não são permitidos ciclos, ou seja, um nó não pode ter como filhos seus nós ancestrais. As árvores podem ser classificadas de acordo com o número de filhos que cada nó pode ter; na figura a seguir temos uma *árvore binária*, na qual cada nó pode ter até dois filhos. Esta estrutura é bastante usada no gerenciamento de bancos de dados:



A declaração do nó base desta árvore segue o seguinte modelo:

```
type
  ApontaNo = ^No ;
  No = record
    valor   : integer ;
    FilhoEsquerdo,
    FilhoDireito : ApontaNo ;
  end ;

var
  raiz : ApontaNo ;
```

A construção e manipulação de estruturas dinâmicas pode se tornar uma tarefa extremamente complexa se o programador não tiver em mente exatamente o que deseja. Além disso, os algoritmos envolvendo estruturas de dados dinâmicas devem ser muito bem projetados e implementados, pois é muito fácil cometer erros na manipulação dos apontadores e com isso invadir áreas de memória reservadas para outras finalidades, correndo o risco de bloquear completamente o sistema.

8.4 Exercícios

1. Escreva uma versão não-recursiva e uma versão recursiva de um procedimento para escrever o conteúdo de uma lista encadeada simples de inteiros.

2. Escreva um procedimento **recursivo** para inserir um elemento no final de uma lista encadeada simples de inteiros. Esse procedimento recebe como parâmetros um apontador do início da lista e o valor a colocar no novo nó. Ele deve se encarregar também da criação do novo nó (comando `new`).

3. Escreva um procedimento para inserir um valor inteiro em uma lista encadeada simples, mantendo sempre a lista em ordem crescente (o novo elemento deve ser inserido na posição adequada para manter a fila em ordem).

4. Escreva um programa Pascal para gerenciar uma lista de encadeamento duplo contendo valores inteiros. O programa deve ter procedimentos para:

   - Escrever a lista na tela.
   - Inserir um elemento no início da lista.
   - Inserir um elemento no final da lista.
   - Remover um elemento do início da lista.
   - Remover um elemento do final da lista.
   - Remover um elemento indicado por um apontador.
   - Ordenar a lista em ordem crescente de valores.
   - Apagar a lista (recursivamente).

5. Escreva um procedimento para construir uma lista de encadeamento duplo de nomes e datas de aniversário. Cada nova entrada na lista deve ser inserida na posição adequada para mantê-la em ordem crescente de datas e nomes (para uma mesma data os nomes devem estar em ordem crescente).

Capítulo 9

Objetos em Pascal

A programação orientada a objetos é um método de programação que busca aproximar a visão do programa da maneira como percebemos e usamos as coisas (objetos reais) que nos cercam. A principal motivação para programar usando objetos é a eterna necessidade de estruturar os programas. O uso de objetos permite produzir programas bem estruturados, modulares, facilitando correções, modificações e a reutilização do código em outros programas. Um bom exemplo de uso da POO é a interface gráfica do *Windows*^{tm}, toda baseada em objetos: janelas, botões, menus, etc.

Embora possa parecer bastante recente, a programação orientada a objetos foi introduzida em 1967 pela linguagem *Simula*, bastante usada para a simulação de sistemas a eventos discretos. No entanto somente nesta década esse conceito foi mais amplamente difundido.

9.1 Objetos, atributos e métodos

Um objeto real é caracterizado por suas propriedades e pelas ações que podemos efetuar sobre ele. Por exemplo, um carro pode ser descrito por suas características (peso, potência, consumo, velocidade, direção, aceleração, etc.) e pelas ações que podemos efetuar sobre ele para conhecer ou alterar seu estado (ligar, desligar, acelerar, frear, ler o velocímetro, etc.).

Em informática, um objeto normalmente pode ser visto como um tipo especial de registro que contém, além dos dados, procedimentos e funções para a manipulação desses dados. Os dados de um objeto são chamados *atributos*, enquanto os procedimentos e funções associados a esse objeto são chamados *métodos*. Para disciplinar a programação e evitar interações indesejáveis, a única forma de acesso aos atributos de um objeto deve ser através de seus métodos.

Segundo os conceitos da orientação a objetos, um programa é um conjunto de objetos que interagem entre si para oferecer a funcionalidade desejada. Os objetos interagem entre si através de ativações de métodos, que permitem alterar ou consultar seus estados. As ativações de métodos também podem ser vistas como "mensagens" circulando entre os objetos.

9.1.1 Linguagens orientadas a objetos

A linguagem mais conhecida para programação orientada a objetos é *C++*, mas não é a única, nem certamente a melhor. Também permitem programar com objetos as linguagens *Objective-C*, *Smalltalk*, *Ada*, *Eiffel*, *Oberon*, etc. Os compiladores Pascal da linha *Turbo-Pascal* (a partir da versão 5.5) também permitem o uso de objetos, embora de forma mais simples e restrita. Recentemente o sistema *Delphi*^{tm} [Swa96] introduziu um ambiente completo de desenvolvimento

de aplicações orientadas a objeto para a plataforma *Windows*^{tm}, usando como linguagem de base o Pascal e oferecendo recursos como programação visual e gerenciamento de bancos de dados.

Três propriedades caracterizam uma linguagem de programação orientada a objetos:

Encapsulamento : capacidade de formar estruturas contendo dados relacionados e métodos para acessar esses dados, construindo assim objetos. Por exemplo, uma linguagem de POO permite criar um objeto *fila de inteiros*, contendo as variáveis necessárias à implementação da fila e os procedimentos e funções necessários para acessar a fila (inserir ou retirar elementos, obter o número de elementos, procurar valores específicos, etc). O encapsulamento nos permite definir precisamente uma *interface* para o objeto, ou seja, um conjunto de métodos como único meio de acesso ao estado do objeto.

Herança : capacidade de definir objetos e usar essas definições para construir novos objetos, que irão herdar características dos antecessores, como atributos e métodos. Uma classe que herda características de outra classe é chamada *classe descendente*, *subclasse*, *classe herdeira* ou *classe filha*. Uma classe herdeira pode redefinir métodos herdados de suas antecessoras, o que permite modificar parte de seu comportamento para melhor adequa-lo ao fim desejado.

Polimorfismo : capacidade de construir métodos genéricos que podem ser aplicados a diversos níveis de uma hierarquia de objetos (um objeto e seus descendentes).

9.1.2 Metodologias de programação

Para construir uma aplicação usando o conceito de objetos não basta conhecer a sintaxe de alguma linguagem de programação orientada a objetos. Para que o produto final possa fazer uso de todos os benefícios dessa técnica de programação, uma metodologia de trabalho deve ser seguida, desde a análise do problema a resolver até o projeto e a implementação do software, incluindo também sua posterior manutenção.

Nos últimos anos diversas metodologias foram propostas para o projeto de aplicações orientadas a objetos, dentre as quais podemos citar a proposta OMT (*Object Modeling Technique*) como sendo a mais difundida [RBP+91]. De acordo com esta metodologia, o desenvolvimento de uma aplicação deve atravessar as seguintes etapas:

1. *Análise*: partindo de uma descrição inicial do problema, deve-se contruir um modelo da situação real, ressaltando suas propriedades mais importantes. Esse modelo deve indicar claramente o que a aplicação deve fazer, e **não** como ela o fará. O modelo de análise não deve conter nenhuma decisão específica à sua implementação.

2. *Projeto do sistema*: aqui devem ser decididos os aspectos de alto nível relativos à arquitetura do sistema. O sistema deve ser estruturado em módulos, de acordo com a análise efetuada e a arquitetura proposta.

3. *Projeto dos objetos*: nesta etapa são construídos os modelos dos objetos para cada módulo definido na etapa anterior. Estes modelos devem conter informações sobre cada objeto, sua implementação e as relações entre eles. O foco principal desta etapa é a definição das estruturas de dados e dos algoritmos necessários para implementar cada classe de objetos.

4. *Implementação*: as classes de objetos e suas relações, definidas na etapa anterior, são finalmente codificadas em uma linguagem de programação orientada a objetos. Esta etapa

deve pesar pouco no desenvolvimento da aplicação, pois as grandes decisões e a estrutura do sistema devem ter sido completamente determinadas durante as fases anteriores.

A metodologia OMT usa três tipos de modelos para descrever um sistema: o *modelo de objetos*, que descreve os objetos do sistema e suas relações, o *modelo dinâmico*, que descreve as interações entre objetos no sistema, e o *modelo funcional*, que descreve as transformações de dados efetuadas no sistema. A descrição completa de um sistema são requer os três modelos.

Uma apresentação mais profunda desta metodologia está fora do escopo deste texto. Os leitores interessados neste tema devem consultar [RBP+91].

9.2 Objetos em Pascal

Programar usando objetos em Pascal[1] é relativamente simples: a extensão a objetos proposta para essa linguagem deriva em grande parte da noção de registro (`record`). Por exemplo, considere o seguinte registro, que representa uma posição na tela do computador:

```
type
  posição = record
    PosX, PosY : integer ;
  end;
```

Podemos usar o tipo `posição` para criar novos tipos. Por exemplo, podemos criar o tipo `ponto`, que guarda a posição e a cor de um ponto na tela:

```
type
  ponto = record
    pos   : posição ;
    cor   : integer ;
  end;
```

Podemos criar uma variável do tipo `posição`:

```
var
  umapos : posição;
```

Para atribuir um valor inicial à variável `umapos` devemos efetuar:

```
umapos.PosX := 17;
umapos.PosY := 42;
```

Entretanto esses comandos só são válidos para essa variável. Podemos torná-los genéricos criando um procedimento para atribuir os valores de `PosX` e `PosY` a variáveis do tipo `posição`:

```
procedure Inicia(var Pos : Posição; X,Y : integer) ;
begin
  Pos.PosX := X ;
  Pos.PosY := Y ;
end;
```

[1] O conteúdo desta seção somente é válido para o compilador *Borland Turbo-Pascal* versão 6.0 ou mais recente. Outros compiladores podem usar outras sintaxes, ou simplesmente não implementar o suporte à programação orientada a objetos.

Essa solução parece razoável, mas ela somente permite inicializar diretamente variáveis do tipo `posição`. Se quisermos inicializar uma variável do tipo `ponto`, deveremos construir um procedimento específico para esse tipo, usando um nome diferente de `inicia`, por exemplo `iniciaponto`.

O uso da programação orientada a objetos permite resolver facilmente e de maneira elegante problemas como o acima apresentado. Antes de mostrar como, vejamos dois conceitos importantes:

Classe : é o modelo de um objeto, descrevendo sua estrutura interna e seus métodos. É uma noção equivalente à noção de tipo, em Pascal standard.

Objeto : é um objeto propriamente dito, que existe na memória do computador durante a execução do programa. Corresponde à noção de variável, em Pascal standard. Assim como uma variável pertence a um determinado tipo, um objeto pertence a uma determinada classe (também costuma-se dizer que um objeto é uma *instância* de uma determinada classe).

A palavra reservada `object` é usada para definir a estrutura interna de uma classe de objetos, ou seja, seus dados e os cabeçalhos (interfaces) dos métodos disponíveis. Em seguida é necessário descrever completamente cada um dos métodos definidos no interior da classe. Vamos então definir uma classe de objetos chamada `posição`:

```
type
  posição = object
    PosX, PosY : integer ;                { atributos               }
    procedure inicia (X,Y : integer) ;    { interface do método     }
  end;

procedure posição.inicia (X,Y : integer) ; { implementação do método }
begin
  PosX := X ;
  PosY := Y ;
end;
```

Observe que no interior da classe de objetos `posição`, declaramos a interface de um método chamado `inicia`, que permite a inicialização de objetos desta classe. A classe `posição` está *encapsulando* os dados e métodos relacionados a uma posição na tela. Uma vez definida a classe `posição`, podemos criar instâncias de objetos dessa classe (o que equivale, em Pascal standard, a criar variáveis de um tipo determinado):

```
var
  umapos : posição ;
```

Para ativar o método `inicia` do objeto `umapos`, basta executar `umapos.inicia (17,42) ;`. Podemos agora definir uma classe de objetos `ponto`, que vai *herdar* características da classe `posição`:

```
type
  ponto = object (posição)
    Cor : integer ;
  end ;
```

A classe `ponto` assim definida herda da classe `posição` todos os seus campos internos (`PosX` e `PosY`) e métodos (`inicia`), de modo que estes não precisam ser declarados novamente. Uma classe pode herdar características de diversas classes, desde que não haja conflitos de nomes. O método `inicia` pode então ser aplicado sem distinção a objetos da classe `posição` ou `ponto`, o que chamamos *polimorfismo*. Para inicializar a posição de uma variável `umponto` do tipo `ponto` basta executar `umponto.inicia(17,42);`. Aqui detectamos um inconveniente: o método `inicia`, definido para a classe `posição`, não atribui valor inicial ao campo `Cor` dos objetos do tipo `ponto`, o que pode causar problemas. Para resolver isso, é possível redefinir o método `inicia` para a classe `ponto` (e seus descendentes), permitindo a atribuição de um valor inicial ao campo `Cor`:

```
type
  ponto = object (posição)
    Cor : integer ;
    procedure inicia (X,Y,C: integer) ;
  end;

procedure ponto.inicia(X,Y,C: integer);
begin
  posição.inicia (X,Y) ;
  Cor := C ;
end;
```

Observe que o nome do método permanece o mesmo (polimorfismo). Desta forma, mudanças nas definições dos métodos não tem conseqüência sobre outros objetos que os utilizam. Além disso, a nova implementação do método `inicia` para a classe `ponto` usa o método herdado da classe `posição` para definir os valores iniciais das coordenadas `PosX` e `PosY`. Isso ajuda a manter a modularidade do código, evitando que o método `ponto.inicia` inicialize diretamente valores próprios da classe `posição`. Além disso, modificações futuras no método `posição.inicia` serão automaticamente incorporadas ao método `ponto.inicia`.

9.3 Campos públicos e privados

Uma boa regra de programação é somente acessar os dados de um objeto através de seus métodos. Isso permite que sejam efetuadas mudanças na estrutura de dados de um objeto sem precisar alterar os objetos que o acessam. Entretanto as definições de classes de objetos apresentadas até agora não impedem que sejam feitos acessos diretos aos dados dos objetos. Por exemplo, nada impediria um programador de acessar e alterar `umponto.posX`. Para evitar acessos indevidos e indesejáveis (e para disciplinar a programação), podem ser definidas áreas públicas e privadas dentro de uma classe de objetos. Para tal, usa-se a palavra reservada `private`, da seguinte forma:

```
type
  nome = object (...)
  definição dos dados e métodos públicos
  private
  definição dos dados e métodos privados
end ;
```

Os campos (dados e métodos) definidos na área pública podem ser acessados diretamente por qualquer outro objeto. Os demais, na área privada, só podem ser acessados pelos métodos do próprio objeto. Um método declarado na área privada de um objeto só pode ser chamado por métodos do mesmo objeto, e por nenhum método externo. Vejamos uma definição da classe de objetos `Posição` utilizando estes conceitos:

```
type
  Posição = object
    procedure inicia  (x,y   : integer) ;
    procedure desloca (dx,dy : integer) ;
    function x : integer ;
    function y : integer ;
    private
    PosX, PosY : integer ;
  end ;

procedure Posição.inicia (x,y : integer) ;
begin
  PosX := x ;
  PosY := y ;
end ;

procedure Posição.desloca (dx,dy : integer) ;
begin
  PosX := PosX + dx ;
  PosY := PosY + dy ;
end ;

function Posição.x : integer ;
begin
  x := PosX ;
end ;
```

9.4 Métodos virtuais

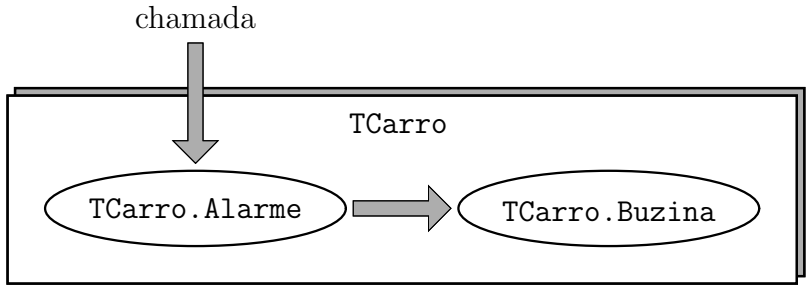
Imagine uma classe de objetos `TCarro`, contendo os métodos `Alarme` e `Buzina`, sendo que o método `Alarme` chama em seu corpo o método `Buzina`. A declaração dessa classe é dada a seguir:

```
type
   TCarro = object
      procedure Buzina ;
      procedure Alarme ;
   end ;

   procedure TCarro.Buzina;
   begin
      writeln ('fom fom') ;
   end ;

   procedure TCarro.Alarme ;
   begin
      Buzina ;
      writeln ('Chamem a policia !') ;
   end ;
```

Uma chamada ao método `TCarro.Alarme` vai gerar a seguinte saída:

```
fom fom
Chamem a policia !
```

Essa saída é gerada pelo seguinte encadeamento de métodos:



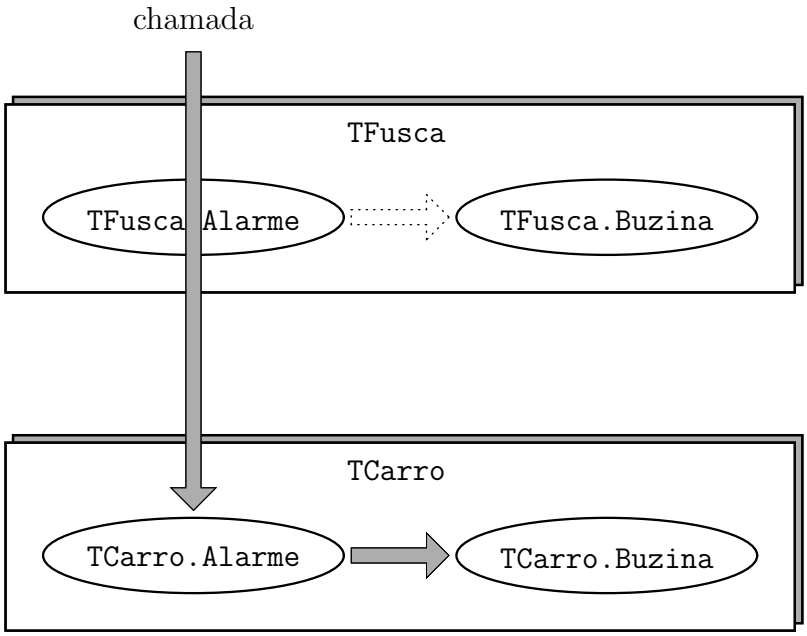
A partir da classe `TCarro` podemos definir uma nova classe `TFusca` que herda as características de `TCarro`. Por razões óbvias, vamos ter de redefinir o método `Buzina` para a classe `TFusca`:

```
type
   TFusca = object (TCarro)
      procedure Buzina ;
   end ;

   procedure TFusca.Buzina ;
   begin
      writeln ('bi bi') ;
   end ;
```

Entretanto, ao ativar o método `TFusca.Alarme` vamos obter a mesma saída anterior:

```
fom fom
Chamem a policia !
```

Esta saída resulta do seguinte encadeamento de chamadas:



Apesar da redefinição, o método `TFusca.Alarme` continua a chamar o método `TCarro.Buzina`, ao invés de `TFusca.Buzina`. Isto se deve ao fato de que a chamada do método `TCarro.Alarme` ao método `TCarro.Buzina` é processada durante a compilação do programa e permanece fixa, constituindo assim os chamados *métodos estáticos*. Para que `TCarro.Alarme` ative `TFusca.Buzina` é necessário descobrir, durante a execução do programa, que objeto ativou `TCarro.Alarme`, e verificar se o mesmo redefiniu o método `Buzina`. Isto pode ser feito através dos *métodos dinâmicos* ou *virtuais*.

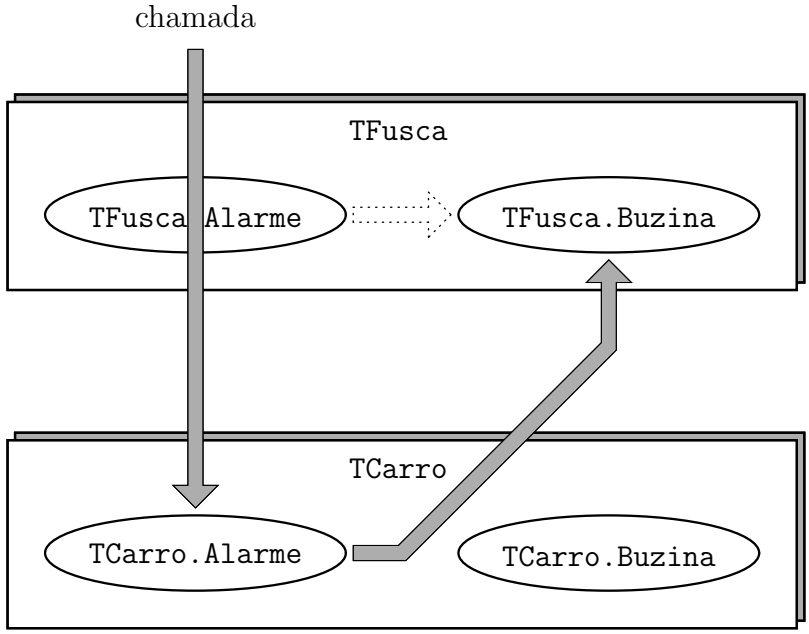
Para definir uma classe de objetos com métodos virtuais, precisamos definir um *método construtor* para essa classe. Esse método, definido pela palavra reservada `constructor`, deve ser chamado antes de qualquer outro método do objeto, uma única vez para cada novo objeto da classe. O corpo desse método pode ser vazio ou conter operações de inicialização dos dados do objeto. Além disso, é necessário indicar quais métodos são passíveis de redefinição nas classes descendentes. Isto é feito através da palavra reservada `virtual`. Aplicando esses conceitos, vejamos como fica a nova definição das classe `TCarro` usando métodos virtuais:

```
type
   TCarro = object
      constructor Inicia ;           { Método construtor desta classe }
      procedure   Buzina ; virtual ; { Este é um método virtual       }
      procedure   Alarme ;
   end ;

   constructor TCarro.Inicia ;
   begin
     { podemos colocar aqui a inicialização do objeto carro }
   end ;

   procedure TCarro.Buzina;
   begin
      writeln ('fom fom') ;
   end ;

   procedure TCarro.Alarme ;
   begin
      Buzina ;
      writeln ('Chamem a policia !') ;
   end ;
```

Para a classe `TFusca` teremos a seguinte implementação:

```
type
   TFusca = object (TCarro)
      procedure Buzina ; virtual ;   { Este é um método virtual }
   end ;

   procedure TFusca.Buzina ;
   begin
      writeln ('bi bi') ;
   end ;
```

Usando os métodos virtuais, o encadeamento das chamadas de métodos corresponde ao desejado:



Com esse encadeamento de métodos obtemos a saída desejada, pois sendo `TCarro.Alarme` um método virtual, ele fará uma chamada ao método `Buzina` do objeto que o invocou, em nosso exemplo o objeto `Fusca`:

```
bi bi
Chamem a policia !
```

Deve-se observar que, caso um método seja declarado como virtual em uma classe, ele permanecerá virtual em todas as suas classes descendentes.

9.5 Objetos e apontadores

Como todas as variáveis, as instâncias de objetos podem ser criadas dinamicamente e referenciadas por apontadores. Para criar e destruir objetos dinâmico podemos usar os comandos `new` e `dispose` da forma habitual:

```
program MeuObjetoDinamico ;
type
  UmObjeto = object
    Nome : string [40] ;
    constructor Init ;
  end ;

  constructor UmObjeto.Init ;
  begin
  end ;

  PtrUmObjeto = ^UmObjeto ;

var
  MeuObjeto = PtrUmObjeto ;

begin
  new (MeuObjeto) ;
  MeuObjeto^.Nome := 'Eu mesmo' ;
  dispose (MeuObjeto) ;
end.
```

No entanto o comando `new` permite outras sintaxes: ele pode ser usado como uma função, recebendo como parâmetro o tipo de objeto a criar e retornando um apontador para o mesmo. Além disso, podemos passar como segundo parâmetro o método de inicialização do objeto, que será então automaticamente ativado logo após a sua criação:

```
MeuObjeto := new (PtrUmObjeto) ;

  ou

MeuObjeto := new (PtrUmObjeto, Init) ;
```

Um dos maiores interesses em usar ponteiros para objetos reside no fato de que **um ponteiro para objetos de uma determinada classe pode também apontar para objetos de qualquer uma de suas classes descendentes** (mas o contrário não é verdadeiro !). Com isso podemos criar apontadores genéricos, capazes de apontar para qualquer objeto dentro de uma determinada hierarquia [Swa91]. Veremos como isso funciona no exemplo da próxima seção.

9.6 Um segundo exemplo

Vamos abordar um exemplo mais complexo e detalhado do uso do conceito de objetos, para ressaltar suas características de herança e polimorfismo, e os benefícios que estas podem trazer para a estruturação dos programas e a reutilização de código.

Neste exemplo vamos esboçar um sistema simples de registro de funcionários de uma empresa. Em relação à forma de pagamento, podemos classificar os funcionários em três grupos: mensalistas (que recebem um salário fixo mensal), horistas (que recebem por hora trabalhada) e comissionados (que recebem um fixo e uma comissão sobre suas vendas).

Como cada funcionário será representado por um objeto, devemos definir uma classe base que representará o modelo básico de empregado, e dela derivar cada uma das classes acima mencionadas. Vejamos agora a definição da classe base:

```
type
  EmpregBase = object
    Nome  : string [40] ;
    Idade : byte ;
    constructor Init ;
    destructor  Done ;
    procedure   Preenche ;       virtual ;
    procedure   Mostra   ;       virtual ;
    function    Salario : real ; virtual ;
  end ;
```

A implementação dos métodos da classe `EmpregBase` toma a seguinte forma:

```
  constructor EmpregBase.Init ;
  begin
  end ;

  destructor EmpregBase.Done ;
  begin
  end ;

  procedure EmpregBase.Preenche ;
  begin
    write ('Nome : ') ;
    readln (Nome) ;
    write ('Idade: ') ;
    readln (Idade) ;
  end ;

  procedure EmpregBase.Mostra ;
  begin
    writeln ('Func: ',Nome,', ', Idade, ' anos, Salario R$ ', Salario:7:2) ;
  end ;

  function EmpregBase.Salario : real ;
  begin
    Salario := 112.00 ; { salário mínimo }
  end ;
```

A partir da definição da classe `EmpregBase` podemos derivar uma classe para os empregados horistas. Além dos campos básicos, esta nova classe possui dois novos atributos: o valor pago por hora e o número de horas trabalhadas. Por esta razão devemos redefinir a implementação dos métodos `Preenche` e `Salario` para a nova classe `EmpregHorista`:

```
type
  EmpregHorista = object (EmpregBase)
    SalHora  : real ;
    NumHoras : integer ;
    procedure Preenche       ; virtual ;
    function  Salario : real ; virtual ;
  end ;

  procedure EmpregHorista.Preenche ;
  begin
    EmpregBase.Preenche ;              { para ler os dados básicos }
    write ('Valor da hora: ') ;
    readln (SalHora) ;
    write ('Horas trabalhadas:' ) ;
    readln (NumHoras) ;
  end ;

  function EmpregHorista.Salario : real ;
  begin
    Salario := SalHora * NumHoras ;
  end ;
```

Observe que o método `Preenche` redefinido faz uso do método original da classe base para a leitura de alguns dados. Da mesma forma, vamos definir uma nova classe para os empregados comissionados:

```
type
  EmpregComissao = object (EmpregBase)
    SalFixo, Vendas, Comissao : real ;
    procedure Preenche       ; virtual ;
    function  Salario : real ; virtual ;
  end ;

  procedure EmpregComissao.Preenche ;
  begin
    EmpregBase.Preenche ;
    write ('Salario fixo: ') ;
    readln (SalFixo) ;
    write ('Valor das vendas: ') ;
    readln (Vendas) ;
    write ('Percentual de comissao: ') ;
    readln (Comissao) ;
  end ;

  function EmpregComissao.Salario : real ;
  begin
    Salario := SalFixo + Comissao * Vendas ;
  end ;
```

Vamos agora criar um vetor de funcionários. Para poder colocar no mesmo vetor objetos das três classes, sem distinção, vamos usar um vetor de apontadores para objetos da classe `EmpregBase` (como vimos, um apontador para uma determinada classe pode também apontar para objetos de classes derivadas desta). Para isso precisamos definir apontadores para todas as classes definidas acima:

```
type
  PtrEmpregBase     = ^EmpregBase ;
  PtrEmpregHorista  = ^EmpregHorista ;
  PtrEmpregComissao = ^EmpregComissao ;

var
   Empregado    : array [1..100] of PtrEmpregBase ;
   NumEmp,i     : integer ;
   TotalSalario : real ;

begin
  NumEmp := 3 ;
  Empregado[1] := new (PtrEmpregBase,     Init) ;
  Empregado[2] := new (PtrEmpregHorista,  Init) ;
  Empregado[3] := new (PtrEmpregComissao, Init) ;

  for i := 1 to NumEmp do
     Empregado[i]^.Preenche ;
  for i := 1 to NumEmp do
     Empregado[i]^.Mostra ;

  TotalSalario := 0 ;
  for i := 1 to NumEmp do
     TotalSalario := TotalSalario + Empregado[i]^.Salario ;
  writeln ('Total de salarios: R$ ', TotalSalario:7:2) ;
end.
```

No laço de cálculo do total de salários podemos observar que a chamada ao método `Salario` de cada objeto se ajusta de maneira transparente a cada tipo de empregado. Da mesma forma, o método `Mostra`, definido somente na classe de base, seleciona o método `Salario` adequado à classe de cada objeto que o ativar.

9.7 Exercícios

1. Escrever uma classe de objetos *fila de inteiros*, com métodos para inicializar a fila, inserir um elemento no final, remover um elemento do início, escrever a fila, retornar o número de elementos da fila e retornar o i-ésimo elemento da fila. Para a implementação da fila use uma lista duplamente encadeada.

2. Criar uma classe de objetos gráficos *figura*, com métodos para entrada de dados (dimensões), apresentação dos dados (incluindo o nome), cálculo da área da figura e de seu perímetro. A partir desta classe básica, derivar classes para círculo, quadrado, retângulo, triângulo, etc, redefinindo os atributos e métodos necessários. O construtor de cada classe deve atribuir um nome ao objeto criado (triângulo, quadrado, etc). A apresentação de uma figura na tela deve seguir o seguinte modelo:

   ```
   Olá, eu sou um ********** de lados ***, *** e ***.
   Minha área vale ****.** e meu perímetro ****.**.
   ```

   Com as classes assim definidas, construa um vetor de apontadores para diferentes tipos de figuras (nos moldes do exemplo da seção 9.6), preencha todos os dados, apresente-os na tela e calcule a área total das figuras.

Referências Bibliográficas

[FE93] A. L. V. Forbellone and H. F. Eberspächer. *Lógica de Programação – A Construção de Algoritmos e Estruturas de Dados*. Editora Makron Books, 1993.

[GL85] A. M. Guimarães and N.A.C. Lages. *Algoritmos e estruturas de dados*. Editora Livros Técnicos e Científicos, 1985.

[MM89] I. Mecler and L. P. Maia. *Programação e lógica com Turbo Pascal*. Editora Campus, 1989.

[O'B92] S. O'Brien. *Turbo-Pascal 6 – Completo e Total*. Editora McGraw Hill, 1992.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Editora Prentice Hall, 1991.

[Sed83] R. Sedgewick. *Algorithms*. Editora Addison-Wesley, 1983.

[Swa91] T. Swan. *Programando em Pascal 7.0 para Windows Borland*. Editora Berkeley do Brasil, 1991.

[Swa96] T. Swan. *Delphi - Bíblia do programador*. Editora Berkeley do Brasil, 1996.

[VSAF86] P. Veloso, C. Santos, P. Azeredo, and A. Furtado. *Estruturas de Dados*. Editora Campus, 1986.